HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

DIPLOMARBEIT

Herr
Alexander Köpke

Betrachtung von Architekturen für komplexe Softwaresysteme im .NET-Umfeld

2012

Fakultät **Elektrotechnik/Informationstechnik**

DIPLOMARBEIT

Betrachtung von Architekturen für komplexe Softwaresysteme im .NET-Umfeld

Autor:
Alexander Köpke

Studiengang:
Informationstechnik (postgradual)

Seminargruppe:
KI09w1

Erstprüfer:
Prof. Dr.-Ing. Wilfried Schubert

Zweitprüfer:
Dipl.-Ing.(FH) Stefan Linke

Mittweida, 01 2012

Bibliografische Angaben

Köpke, Alexander: Betrachtung von Architekturen für komplexe Softwaresysteme im .NET-Umfeld, 141 Seiten, 32 Abbildungen, Hochschule Mittweida (FH), Fakultät Elektrotechnik/Informationstechnik

Diplomarbeit, 2012

Referat

Komplexe Softwaresysteme haben in der Regel ähnliche Architekturen (Schichtenmodell, Services, dynamisch ladbare Features, Einsatz von GoF Pattern). Einige Architekturmittel können in Verbindung mit .NET Technologien zu Problemen im Speicherverbrauch und der Laufzeit führen. Diese gilt es aufzuzeigen und Lösungen zu erarbeiten. Neben den Auswirkungen der Architektur auf das Gesamtsystem sollen unter anderem auch die folgenden Themen betrachtet werden, da sie in jedem .NET Projekt eine Rolle spielen:

- Garbage Collection (Large Object Heap)
- Lebenszyklus von .NET Objekten (IDisposable, Finalizer)
- Dynamische Codeerzeugung mittels CodeDom
- Laden von Features via Reflection
- Wachstumsverhalten/Laufzeitverhalten von Collections

I. Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis II
Tabellenverzeichnis III
Abkürzungsverzeichnis IV
Vorwort V
1 Einleitung 1
1.1 Betrachtung des Themas 1
1.2 Aufgabenstellung 2
1.3 Abgrenzung des Themas 3
2 Notwendige Grundlagen 4
2.1 Architekturstile 4
2.1.1 Schichtenmodell 4
2.1.2 Objektorientierung 7
2.1.3 Aspektorientierung 8
2.1.4 Dienstorientierung 10
2.2 Architekturmuster 11
2.2.1 Domänen-Modelle 11
2.2.2 Dynamisch ladbare Funktionalitäten 14
2.2.3 Model-View-ViewModel 17
2.2.4 Verwalteter und Nativer Quelltext 22
2.3 Entwurfsmuster 28
2.3.1 Gang of Four Pattern 29
2.4 .NET Framework 31
2.4.1 Speicherverwaltung 31
2.4.2 Wachstumsverhalten von Listen 34
2.4.3 Eventsystem 35
2.4.4 Lebenszyklus von Objekten 37
2.4.5 Zeichenketten 38

2.4.6 Nicht verwaltete Ressourcen ... 43
2.5 Probleme komplexer Softwaresysteme ... 46
2.5.1 Massenoperationen ... 46
2.5.2 Metadateien ... 48
3 Unterscheidung von komplexen Softwarearchitekturen ... 49
3.1 Offene und geschlossene Architekturen ... 49
3.2 Unterscheidung nach der Programmiersprache ... 51
3.3 Unterscheidung nach dem Aufbau ... 51
3.3.1 Multithread ... 52
3.3.2 Multiprozess ... 52
3.3.3 Client-Server ... 53
3.3.4 Verteilt ... 54
3.4 Nach der Oberflächentechnologie ... 54
3.4.1 Windows Forms ... 54
3.4.2 Windows Presentation Foundation (WPF) ... 56
3.4.3 METRO ... 60
3.4.4 Silverlight ... 63
4 Favorisierter Ansatz ... 68
4.1 Allgemein ... 68
4.2 Architekturaufbau ... 69
4.3 Oberflächentechnologie ... 70
5 Auswirkungen der Architektur im .NET-Umfeld ... 71
5.1 Allgemeine Probleme in großen Projekten ... 71
5.1.1 Schwergewichtige Objekte ... 71
5.1.2 Abhängigkeiten von Objekten ... 73
5.1.3 Navigation über das Modell ... 76
5.1.4 Fehlersuche bei deskriptiven Ansatz ... 77
5.2 Spezielle Probleme im .NET-Umfeld ... 79
5.2.1 Interoperabilität ... 79
5.2.2 Obfuskation und Meta-Basierte Reflektion ... 80
5.2.3 Externe und interne Schnittstellen ... 82

5.2.4 Reflektion und Quelltextgenerierung 83
6 Zusammenfassung 85
6.1 Darstellung der wesentlichen Ergebnisse 85
6.2 Erkenntnisse 90
6.3 Ausblick 92
A Anhang 93
A.1 Quellcodes 93
A.1.1 Anwendung entworfen nach MVVM 93
View 93
ViewModel 95
Model 100
A.1.2 155MB für Out Of Memory 102
A.1.3 Einfache LOH freie Collection 104
A.1.4 String interning ohne dauerhaften Speicherverbrauch 111
A.1.5 Graph-Plotter WinForms Usercontrol 112
A.1.6 Graph-Plotter WPF Usercontrol 116
XAML der Anwendung 116
Programmcode des User Controls 116
A.1.7 Graph-Plotter METRO Usercontrol 122
XAML der Anwendung 122
XAML des User Controls 123
Programmcode des User Controls 123
A.1.8 Graph-Plotter Silverlight Usercontrol 128
A.2 Tutorial: Visual Studio für Entwicklung von verwalteten und nativen Quelltext einrichten 129
A.3 Spezialisierung Domänen-Modell 131
A.3.1 Package - Veranstaltung 131
A.3.2 Package - Spieler 134
A.3.3 Package - Disziplin 135
A.3.4 Package - Lauf 136
Literaturverzeichnis 137
Glossar 138

II. Abbildungsverzeichnis

2.1 Vereinfachtes Schichtenmodell einer Anwendung 5

2.2 Funktionsweise von PostSharp (Bildquelle: PostSharp) 10

2.3 Domänen-Modell einer Laufsport-Veranstaltung 13

2.4 Überblick des Aufbaus des Model-View-ViewModels 18

2.5 WPF Anwendung mit Suchfunktion in einer Aufzählung von Autoren 21

3.1 Klassendiagramm - Oberflächenänderung persistieren 50

3.2 Sequenzdiagramm - Oberflächenänderung persistieren (geschlossene Architektur) ... 50

3.3 Sequenzdiagramm - Oberflächen Änderung persistieren mittels offener Architektur ... 50

3.4 Anwendung mit einer Windows Forms Oberfläche 55

3.5 WinForms-Anwendung mit selbst gebautem Graphen-Plotter 56

3.6 Mit WPF implementierter WYSIWYG-Blog-Editor 57

3.7 WPF-Anwendung mit selbst gebauten Graphen-Plotter 60

3.8 Windows8 Tablet-Desktop (METRO-Desktop) 61

3.9 Überblick die Plattformen und Werkzeuge in Windows 8 (Bildquelle: Microsoft) 62

3.10 Als METRO app implementierter Feedreader 63

3.11 Als METRO app implementierter Graph-Plotter 64

3.12 Architektur von Silverlight, Bildquelle: Microsoft 66

3.13 Silverlight Anwendung mit selbst gebautem Graph-Plotter 66

5.1 Beispielhafte Darstellung des Projektbaumes einer Anwendung 76

6.1 Übersicht der Probleme des .NET Frameworks mit Lösungsideen 85

A.1 Unterschiede in der Darstellung eines Silverlight UserControl zwischen Browser und Designer 129

A.2 Solution Navigator, zeigt eine Solution mit nativem und verwaltetem Projekten. 130

A.3 Starten des Konfigurationsmanagers. 130

A.4 Konfigurationsmanagers, alle Projekte sollten die gleiche Zielplattform haben. 130

A.5 Aktivieren der DEBUG Einstellung für native Projekte. 131

A.6 Aktivieren der DEBUG Einstellung für verwaltete Projekte. 131

A.7 Aktivieren der Einstellung "Enable unmanged code debugging". 132

A.8 Aktivieren der Einstellung "Enable unmanged code debugging". 132
A.9 Mögliche Realisierung des Packages "Veranstaltung aus einem Domänen-Modell" . . . 133
A.10 Mögliche Realisierung des Packages "Spieler aus einem Domänen-Modell" 134
A.11 Mögliche Realisierung des Packages "Disziplin aus einem Domänen-Modell" 135
A.12 Mögliche Realisierung des Packages "Lauf aus einem Domänen-Modell" 136

III. Tabellenverzeichnis

2.1 Übersicht und Gegenüberstellung von Lösungsvorlagen für Architekturen 4

2.2 Auflistung aller Strings im Speicher des "Hallo Welt"-Programms 40

2.3 Auflistung aller Strings im Speicher nach Speicher-Optimierung des "Hallo Welt"-Programms 41

2.4 Auflistung aller Strings im Speicher nach Laufzeit-Optimierung des "Hallo Welt"-Programms 42

3.1 Vor- und Nachteile von offener und geschlossener Architektur 51

6.1 Leitfaden für den Softwarearchitektur-Entwurf 89

IV. Abkürzungsverzeichnis

API ..... Application Programming Interface

CASE ..... Computer-Aided Software Engineering

CLI ..... Common Language Infrastructure

CLR ..... Common Language Runtime

CLS ..... Common Language Specification

COM ..... Component Object Model

CSS ..... Cascading Style Sheet

CTS ..... Common Type System

DBMS ..... Database Mangement System

DNS ..... Domain Name Service

GC ..... Garbage Collector

GDI ..... Graphic Device Interface

GoF ..... Gang of Four

GRASP ..... General Responsibility Assignment Software Patterns

ID ..... Identifier

IP ..... Internet Protocol

IPC ..... Inter Process Communication

JIT ..... Just In Time

LOH ..... Large Object Heap

LSB ..... Least Significan Bit

MAC ..... Media Access Control

MSB ..... Most Significan Bit

MTA ..... Multi Thread Apartment

MVC ..... Model View Controller

MVVM ..... Model View ModelView

OS ..... Operating System

RDD	Responsiblity-Driven Design
RIA	Rich Internet Application
SDK	Software Development Kit
SQL	Structured Query Language
STA	Single Thread Apartment
TPL	Task Parallel Library
UI	User Interface
UML	Unified Modelling Language
WCF	Windows Communication Foundation
WinRT	Windows Runtime
WPF	Windows Presentation Foundation
WWW	World Wide Web
WYSIWYG	What You See Is What You Get
XAML	Extensible Application Markup Language
XBA	XML Browser Application
XML	Extensible Markup Language
z.B.	zum Beispiel

Listings

2.1 Logging in einer BusinessLogic Klasse über Aspektorientierung. 9
2.2 Einfache Implementierung eines Log-Aspekts. 9
2.3 Über Reflektion realisiertes Tracing. 15
2.4 Reflektion in Verbindung mit AppDomains. 16
2.5 Anbindung der View an das ViewModel über Properties 18
2.6 Informierung der View über Änderungen durch das ViewModel mit INotifyPropertyChange 19
2.7 Ansteuerung der Funktionalitäten des ViewModels durch die View über Commands 20
2.8 Verwendung einer WinAPI Funktion über P/Invoke. 22
2.9 Interface-Beschreibung einer COM-Schnittstelle zur Ermittlung des Computernamens. 23
2.10 Aus Typdatei mittels tlbimp.exe erzeugter C# Zugriffscode. 24
2.11 Verwendung der von tlbimp.exe erzeugten C# Zugriffsklassen. 25
2.12 Einfache C# Klasse, welche die loakle IP-Adresse des Rechners zurück gibt. 26
2.13 Hosting der CLR aus C++, zum Auslesen der lokalen IP-Adresse. 27
2.14 Methode zum Auslesen der aktuellen Heap-Generation eines Objektes. 33
2.15 Erzwingen einer Garbage Collection. 33
2.16 Beispiel eines Objektes, das auf dem LOH gespeichert wird. 34
2.17 Richtiges Anlegen einer Liste, bei abschätzbarer Anzahl der Elemente. 35
2.18 Anwendung für die Erhaltung alle Objekte durch Eventanmeldungen. 35
2.19 Schlechtes Beispiel für verarbeitung von Zeichenketten. 39
2.20 Zeichenkettenverarbeitung mit weniger Speicherverbrauch. 40
2.21 Schnellste Variante zur Verarbeitung von Zeichenketten. 41
2.22 Verwendung von String.Interning um Vergleiche von Zeichenketten zu beschleunigen. 43
2.23 Cache zum Verwalten von Bitmap-Objekten. 44
2.24 Cache zum Verwalten von Bitmap-Wrapper-Objekten. 45
3.1 Definition einer Oberfläche über den deklarativen XAML-Ansatz. 57
5.1 Event Hantierung ohne angemeldete Objekte am Leben zu halten. 73
5.2 META-Beschreibung zu Aufbau einer Oberfläche mittels Reflektion. 77
5.3 Dekompiliertes Programm ohne Obfuskation 80
5.4 Dekompiliertes Programm mit einfacher Obfuskation. 81
A.1 MVVM Beispiel: XAML der View 93
A.2 MVVM Beispiel: Codebehind der View 94
A.3 MVVM Beispiel: mögliche Implementierung des ViewModels 95
A.4 MVVM Beispiel: Element als Teil des ViewModels 97
A.5 MVVM Beispiel: mögliche Implementierung eines Commands. 99
A.6 MVVM Beispiel: Simulation eines Models 100
A.7 MVVM Beispiel: Element eines Models 102

A.8 Ende des Speichers nach 155 MB. 102
A.9 Niemals auf den LOH gespeicherte doppelt verkettete Liste. 104
A.10 String Interning ohne dauerhaften Speicherverbrauch 111
A.11 Graph-Plotter als WinForms Oberflächenelement 112
A.12 XAML einer WPF-Anwendung (mit Graph-Plotters als WPF-Oberflächenelement).116
A.13 Codebehind eines Graph-Plotters als WPF Oberflächenelement 116
A.14 XAML einer METRO-Anwendung (mit Graph-Plotter als METRO Oberflächenelement). 122
A.15 XAML eines Graph-Plotter als METRO-Oberflächenelement. 123
A.16 Codebehind eines Graph-Plotters als METRO-Oberflächenelement. 123
A.17 Codebehind eines Graph-Plotters als Silverlight-Oberflächenelement. 129

V. Vorwort

Die vorliegende Diplomarbeit wurde zwischen September 2011 und Januar 2012 an der Fakultät Mathematik, Naturwissenschaft und Informatik unter der Leitung von Herr Prof. Dr. Ing. Wilfried Schubert angefertigt. Als Basis der Arbeit zählt die Praxiserfahrung in der Entwicklung von Software bei der Siemens AG in Nürnberg. Fachlicher Betreuer seitens der Siemens AG ist Herr Dipl.-Ing. (FH) Stefan Linke.
Mein besonderer Dank gilt Herrn Prof. Dr. Ing. Wilfried Schubert und Herrn Dipl.-Ing. (FH) Stefan Linke, sowohl für die Betreuung der Arbeit als auch für die Übernahme der Erst- und Zweitkorrektur.

Mein Dank gilt auch der Siemens AG, die es mir ermöglicht, eine Diplomarbeit zu erstellen, die nicht für die Veröffentlichung gesperrt werden muss und somit für akademische Zwecke genutzt werden kann.

Nürnberg im Januar 2012
Alexander Köpke

1 Einleitung

1.1 Betrachtung des Themas

Unter komplexen Softwaresystemen versteht man IT-Großprojekte (Projektdauer über 2 Jahre, Projektbudget über 8 Millionen €, mehr als 250 Projektbeteiligte). Diese Projekte haben ein hohes Potenzial zu scheitern (aus [Pavlik]):

- "46% der IT-Vorhaben haben zumindest teilweise nicht die Wünsche und Anforderungen der Auftraggeber erfüllt. Jedes fünfte Projekt ist ein Totalausfall.
  ("Chaos Report" der Standish Group 2006)
- Nur knapp die Hälfte aller IT-Vorhaben der vergangenen drei Jahre war erfolgreich. Sie dauerten entweder länger als geplant, kosteten wesentlich mehr oder es kam am Ende ein anderes Ergebnis heraus. Andere Projekte mussten sogar abgebrochen werden, wobei in der Regel viel Geld in den Sand gesetzt wird.
  (Studie der Technischen Universität München)
- 20% aller IT-Projekte werden abgebrochen; jedes zweite dauert länger oder wird teurer als geplant. Die Wahrscheinlichkeit des Scheiterns steigt mit der Dauer und Komplexität von Projekten.
  (Studie "Projekte mit Launch Management auf Kurs halten. Warum IT-Großprojekte häufig kentern und Projekterfolg kein Glücksspiel ist", Roland Berger Strategy Consultants, 2008)
- Nur 16% der untersuchten IT-Projekte können als erfolgreich eingestuft werden.
  (Studie der Universität Oxford 2003)"

Das Fehlschlagen von Projekten hat viele Ursachen, von denen einige dem Management oder der Projektleitung zuzuordnen sind. Dazu gehören:

- Mangelndes Anforderungsmanagement
- Ungünstige Projektstruktur bzw. Zusammenstellung von Projektteams
- Unzureichende Projektplanung und -steuerung

Den Softwarearchitekten und Softwareentwicklern sind die folgenden zuordenbar:

- Komplexität des Projektes wird unterschätzt
- Implementierung auf Basis von "Sunny-Day-Szenarien"
- Einsatz von nicht ausreichend evaluierten Technologien

Mit Implementierung auf Basis von "Sunny-Day-Szenarien" ist gemeint, dass Verantwortliche bei der Entwicklung von Funktionalitäten oft von einfachen (Gut-) Fällen ausgehen. Dadurch werden meistens die schwerwiegenden Probleme erst kurz vor Freigabe gefunden. Aktuelle Softwareentwicklungsmethoden (z.B. Agile Entwicklung) verlangen, dass man iterativ arbeiten soll. Dies bedeutet Probleme erst lösen wenn sie entstehen. Doch dies ist nur möglich, wenn man sich vorher nicht die Lösungswege verbaut hat. Wenn also ein Fehler in einer viel benutzten Basiskomponente liegt, ist der Aufwand für Anpassungen um ein Vielfaches höher. Das schlimmste anzunehmende Problem ist es, wenn der Fehler in der Architektur selbst begründet liegt.
Zum "Einsatz von nicht ausreichend evaluierten Technologien" schreibt [Pavlik]:

> "Im Rahmen von IT-Großprojekten werden oft neue Technologien eingesetzt, die erst im Projekt entwickelt bzw. ausgereift werden. Diese Innovationen werden eingesetzt, um First-Mover-Vorteile zu nutzen. Sind diese Technologien oder auch nur Teile davon jedoch noch nicht ausreichend durchdacht, erhöhen sie die Komplexität des Projektes immer mehr und bereiten zusätzliche Probleme. Dadurch entstehen Zusatzaufwände für die Entwicklung und Integration.
>
> In der Softwareentwicklung bestehen 70% einer Applikation heute schon aus Standardkomponenten. Die zwingenden Gründe für eine Neuentwicklung sind nur noch in den wenigsten Fällen vorhanden. Solange Neuentwicklungen nicht unbedingt nötig sind, sollte auf ausgereifte Technologien und bewährte Standards gesetzt werden. Falls Sie neue Technologien zum Einsatz bringen, gilt es diese sorgfältig zu bewerten und zu planen. Des größeren Aufwandes muss man sich voll und ganz bewusst sein."

Dabei muss es nicht um die Entscheidung gehen, ob man ein Projekt in C++ mit MFC, C++ mit Qt, Java oder .NET realisiert. Es geht auch darum, wie sich Komponenten verständigen: Ob ein O/R Mapper verwendet wird und wenn ja, dann welcher. Ob alle Möglichkeiten einer Technologie verwendet werden dürfen. All diese Entscheidungen trifft der Softwarearchitekt. Er trägt maßgeblich zum Erfolg oder Misserfolg eines Projektes bei.

1.2 Aufgabenstellung

Diese Arbeit soll das Wissen vermitteln, dass beim Entwurf der Softwarearchitektur für komplexe Softwaresysteme notwendig ist. Dafür werden wichtige Punkte der Architektur und die relevanten Funktionalitäten des .NET Frameworks im Kapitel 2 beschrieben. Im Kapitel 3 wird die Vielzahl der möglichen "Kernentscheidungen" mit ihren Vor- und Nachteilen aufgezeigt. Kapitel 4 geht auf die möglichen "Kernentscheidungen" ein und diskutiert welche am besten geeignet sind, um somit das "Bild" eines konkreten Soft-

waresystems zu geben. Im Rahmen des Kapitels 5 wird für dieses Softwaresystem auf die Besonderheiten des .NET Frameworks eingegangen. Diese Arbeit soll einige Zusammenhänge zwischen Architekturentscheidungen und der Realisierungstechnologie aufzeigen. Als Abschluss gibt das Kapitel 6 einen Überblick über die Ergebnisse.
Im Kontext dieser Arbeit werden folgende Thesen betrachtet:

1. Die eingesetzte Technologie, beispielsweise .NET, beeinflusst die Softwarearchitektur
2. Entscheidungen in der Softwarearchitektur führen in großen Anwendungen zu mehr Problemen als in kleineren Anwendungen
3. Automatische Speicherverwaltung ermöglicht und vereinfacht die Realisierung von großen Softwareprojekten
4. .NET Anwendungen erzielen eine hohe Portierbarkeit durch Abstraktion des Betriebssystems

1.3 Abgrenzung des Themas

Diese Diplomarbeit erhebt nicht den Anspruch, das gesammelte Wissen, das benötigt wird, um komplexe Softwaresysteme mit .NET erfolgreich umzusetzen, darzustellen. Dies würde den Rahmen dieser Arbeit sprengen. Diese Arbeit ist somit weder ein komplettes Architekturbuch noch ein komplettes .NET Buch. Die Auswahl der vorgestellten Architekturmittel und der .NET Framework-Funktionalitäten wurde vom Autor nach Relevanz für Softwaresysteme getroffen, die aus dem Arbeitsumfeld des Autors stammen. Alle Besonderheiten des .NET Frameworks beziehen sich auf die Common Language Runtime (CLR) V2.0 und können von nachfolgenden CLR Versionen abweichen. Es wird vorausgesetzt, dass der Leser bereits grundlegende Kenntnisse über Programmierung mitbringt. Insbesondere ist eine grundlegende Vertrautheit mit C# und dem .NET Framework notwendig.

2 Notwendige Grundlagen

Dieses Kapitel erklärt grundlegende Theorien, um mögliche "Probleme im .NET-Umfeld" zu erkennen. Viele konkrete Beispiele aus dem .NET-Umfeld, sind in diesem Kapitel enthalten. Dies liegt an der Tatsache, dass eine Software-Architektur immer nur im Kontext der verwendeten Technologie bewertet werden kann. Sollte dem Architekten die Problematik der eingesetzten Technologie nicht bekannt sein, werden in der Folge häufig wichtige Themen in der Architektur nicht vorgesehen bzw. beachtet.

Tabelle 2.1 zeigt die Übersicht und Gegenüberstellung von Lösungsvorlagen für Architekturen nach Torsten Posch (aus [Posch, Seite 207]). In Anlehnung daran wurde diese dreistufige Hierarchie auch in diesem Kapitel zur Gliederung verwendet.

Bezeichnung	Zweck	Beispiele
Architekturstil	Globale Prinzipien zur Strukturierung	Pipes and Filters, Schichten, Blackboard
Architekturmuster	Adressieren querschnittlicher Aspekte	Threads oder Prozesse (für Nebenläufigkeit)
Entwurfsmuster	Lokale Anwendung von bewährtem Entwurfswissen	Observer, Abstract Factory, Singleton

Tabelle 2.1: Übersicht und Gegenüberstellung von Lösungsvorlagen für Architekturen

2.1 Architekturstile

In diesem Kapitel werden einige grundlegende Prinzipien zur Strukturierung der Softwarearchitektur vorgestellt.

2.1.1 Schichtenmodell

Kurze und zutreffende Beschreibung für Schichten: nach [Buschmann, Seite 31]:

> "The *Layers* architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is a particular level of abstraction."

Etwas ausführlicher und mit Beispiel formuliert bedeutet das (aus [Fowler03, Seite 31]):

> "Die Schichtenbildung gehört zu den gebräuchlichsten Techniken, die Software-Designer verwenden, um ein kompliziertes Software-System zu

> zerlegen. Beispielsweise finden Sie bei Maschinenarchitekturen Schichten, die von einer Programmiersprache mit Betriebssystemaufrufen über Gerätetreiber und CPU-Anweisungen bis zu Logikschaltungen in einzelnen Chips reichen. Bei Netzwerken bildet FTP eine Schicht über TCP, das seinerseits auf IP aufsetzt, das wiederum über der Ethernet-Schicht liegt.
> Bei einem Schichtenmodell eines Software-Systems liegen die Haupt-Unter-Systeme wie die Schichten eines Kuchens übereinander. In diesem Schema verwendet die jeweils höhere Schicht verschiedene Dienste, die von den jeweils tieferen Schichten zur Verfügung gestellt werden, während umgekehrt eine tiefere Schicht nichts von den höheren Schichten weiß. Darüber hinaus verbirgt jede Schicht normalerweise die unter ihr liegenden Schichten vor den über ihr liegenden Schichten. Beispielsweise nutzt Schicht 4 die Dienste von Schicht 3, die ihrerseits auf die Dienste von Schicht 2 zugreift; aber Schicht 4 weiß nichts von Schicht 2. Nicht alle Schichtenmodelle sind in dieser Weise transparent, doch die meisten wenden dieses Prinzip zumindest teilweise an."

Durch die Einteilung von Systemen in Schichten wird die Wiederverwendbarkeit erhöht, da Abhängigkeiten reduziert werden. Abbildung 2.1 zeigt ein einfaches Schichtenmodell.

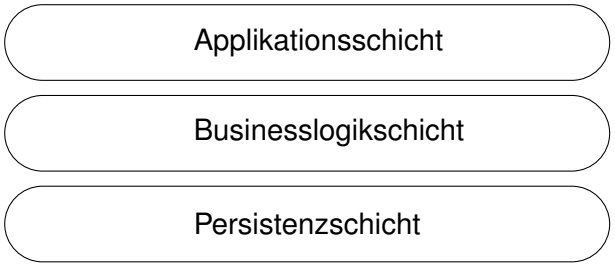

Abbildung 2.1: Vereinfachtes Schichtenmodell einer Anwendung

Die **Applikationsschicht (Application layer)** beinhaltet die gesamt Logik zur Anzeige von Daten und Benutzerinteraktionen.
Die **Businesslogikschicht (Businesslogic layer)** enthält die komplette Geschäftslogik unter anderem:

- Allgemeine Geschäftsregeln
- Spezielle Geschäftsregeln
- Validierung von Daten aus anderen Schichten

Die **Persistenzschicht (Data layer)** ist verantwortlich für das Lesen und Schreiben von Objekten. Diese könnte zum Beispiel als O/R Mapper einer Datenbank realisiert sein.

Es gibt verschiedenste Anwendungen für das Schichten-Modell, wie z.B. das OSI 7-Layer Modell, oder das .NET Framework selber. Programme, welche mittels .NET Framework erstellt sind, werden nicht direkt für eine Zielplattform kompiliert. Statt der Erzeugung des nativen Maschinencodes entsteht sogenannter *IL-Code (Zwischensprachen-Code)*. Zwischencode wird bei .NET vor der Ausführung mittels Just in time Compiler (JIT-Compiler) in Maschinencode übersetzt. Im Gegensatz dazu stehen Java oder Perl, welche den Zwischencode in virtuellen Maschinen interpretieren. Die virtuellen Maschinen werden jeweils auf die Zielplattform portiert. Das entsprechende Framework und der JIT-Compiler sorgen dafür, dass .NET Programme nicht angepasst werden müssen um auf unterschiedlichen Systemen zu laufen. Diese Plattformunabhängigkeit kann in C++ nur durch Verwendung eines entsprechenden Frameworks (wie Qt) oder bedingter Kompilierung (Einsatz von #ifdef) erreicht werden.
Des weiteren existieren Implementierungen vom .NET Framework für unterschiedliche Betriebssysteme[1] und Rechnerarchtekturen (32-Bit, 64-Bit).
Vorteile des Schichtenmodells sind:

- Es ist möglich einzelne Schichten zu verstehen, ohne die unterlagerten Schichten kennen zu müssen. Um den Aufbau der Oberfläche eines Programmes zu verstehen, ist die Kenntnis über den Aufbau der Persistenzschicht irrelevant. Die Persistenzschicht kann Daten in XML speichern, in einer Datenbank ablegen oder ein proprietäres Dateiformat verwenden. Die Wahl ist unabhängig von anderen Schichten, sie wird nur von nichtfunktionalen Anforderungen beeinflusst.
- Schichten können als Schnittstellen (Interfaces) angesehen werden. Schichten können nahezu unabhängig voneinander entwickelt werden und sind austauschbar. Bei einem Messaging-System für Distributed Computing ist nur relevant, dass die unterlagerten Schichten die Daten von einem Rechner zu einem anderen übertragen. Ob die Datenübertragung per WLAN, Kupferdoppelader oder Glasfaserkabel erfolgt ist dabei irrelevant.
- Die Aussage ([Starke, Seite 46]):

  > "Wir wissen, dass Ihnen in Ihrer Schulzeit das Abschreiben streng verboten wurde; uns übrigens auch. Wir plädieren an dieser Stelle dafür, das Abschreiben in Software-Architektur zur Tugend zu erklären."

  zeigt deutlich, dass Wiederverwendung die Komplexität in Softwaresystemen reduziert. Eine besondere Stärke des Schichtensystems ist, dass nicht jede Komponente ihre Übertragungsschicht erneut entwickeln muss, sondern eine bereits implementierte und getestete Schicht verwenden kann. Zudem können Schichten auch in neuen Systemen wiederverwendet werden.

Nachteile des Schichtenmodells:

[1] Microsoft stellt das .NET Framework für verschiedene Windows Versionen bereit. Um .NET Programme unter Unix, Linux oder Mac OS laufen zu lassen, wird das Mono Framework benötigt.

- Kaskadierende Änderungen: Das Schichtenmodell funktioniert nur soweit, wie Anforderungen an einzelne Schichten realisierbar sind. Sollte sich während der Implementierung herausstellen, dass mit gängigen Technologien die Anforderungen an eine Schicht nicht erfüllbar sind, werden Änderungen durch das Schichtenmodell nicht mehr abgefangen. Beispielsweise würde das Umstellen der Anfrage an eine Persistenzschicht, welche das Lesen oder Schreiben ganzer Blöcke verlangt, eine Schnittstellenänderung darstellen und somit auch andere Schichten betreffen.
- Schlechteres Laufzeitverhalten: Daten müssen von jeder Schicht auf Plausibilität geprüft werden - dies sorgt für eine hohe Sicherheit. Für diese Sicherheit muss eine längere Laufzeit akzeptiert werden.
- Eine saubere Architektur (strikte Schichtentrennung) kann durch nichtfunktionale Anforderungen, wie Performance, aufgeweicht werden. Ein Beispiel dafür ist eine Applikationsschicht, die Daten direkt über eine SQL-Abfrage aus einer Persistenzschicht ausliest. In diesem Fall wird eine direkte Abhängigkeit der Schichten geschaffen, was sich negativ auf Pflege, Testbarkeit und die Komplexität des Systems auswirkt. Laufzeit Optimierungen wie Dirty-Flags oder Caches in bestimmten Schichten werden somit umgangen. Dies wirkt sich wiederrum nachteilig auf die Performance des Gesamtsystems aus. Es ist eine signifikante Architekturentscheidung die Schichtentrennung aufzugeben und sollte mit großer Sorgfalt geprüft werden.

2.1.2 Objektorientierung

Torsten Posch beschreibt die Objektorientierung wie folgt (aus [Posch, Seite 212]):

> "Bei diesem Stil werden Architekturen aus kooperierenden Objekten zusammengesetzt. Objekte kapseln dabei Zustandsvariablen und Verhalten. Die Zusammenarbeit zwischen Objekten geschieht durch Zusendung von Nachrichten bzw. durch den Aufruf von Funktionen des empfangenden Objekts. Die Schnittstelle von Objekten beschreibt dabei die möglichen Nachrichten bzw. Funktionen mit ihren Parametern. Objekte sind passiv (d.h. nur von außen auferufen) oder aktiv (d.h. mit einem Eigenleben wie einem eigenen Prozess oder Thread)."

Objektorientierte Softwareentwicklung ist seit Jahren erprobt und eignet sich hervorragend dazu, komplexe Probleme zu lösen. Die größte Kritik an der objektorientierten Programmierung ist, dass sie nicht von vornherein parallelisierbar ist. Dadurch, dass Klassen interne Variablen besitzen können und Änderungen dieser Variablen Auswirkungen auf die Ausführung von Methoden haben können, ist Disziplin bei der Implementierung notwendig. Bei funktionalen Programmiersprachen (Haskell, F#, JavaScript) ist es verpflichtend, dass bei gleichen Eingabeparametern dasselbe Resultat als Rück-

gabewert von den Methoden zurückgegeben wird. Das bedeutet nicht, dass Parallelität mit objektorientierter Programmierung nicht möglich ist - sie wird nur nicht direkt unterstützt. Dies wird von unzähligen Programmen, die produktiv im Einsatz sind, bewiesen. Mit Vererbung, Polymorphie sowie Schnittstellen, bietet Objektorientierung eine Vielzahl von möglichen Ansätzen Probleme zu lösen.
Objektorientierte Programmiersprachen sind ein wichtiger technischer Aspekt. Doch die Objektorientierung unterstützt den Softwarearchitekten bereits beim Entwurf von Software. Der Entwurf behandelt Punkte wie zum Beispiel:

- Welche Klassen/Schnittstellen werden benötigt?
- Was ist die konkrete Aufgabe einer Klasse?
- Welche Klassen müssen miteinander interagieren?
- Wie erfolgt die Interaktion?

Eine Hilfe, um diese Fragen zu klären, sind die **General Responsibility Assignment Software Patterns (GRASP)**. GRASP Regeln sind bekannte Regeln für Zuständigkeitszuweisungen von Objekten, welche von Craig Larman ([Larman]) systematisiert wurden. Nachfolgend zwei wichtige GRAPS für den objektorientierten Entwurf:

Hohe Kohäsion (high cohesion) beschreibt die Verwandtschaft und Fokussierung von Aufgaben innerhalb eines Elementes. Elemente mit einem Fokus auf, im Idealfall eine, bestimmte Aufgabe, welche thematisch verwand sind, besitzen eine hohe Kohäsion. Klassen mit einer niedrigen Kohäsion tragen Verantwortung für verschiedene Funktionalitäten. In der Praxis sind solche Klassen häufig schlecht wartbar und nicht wiederverwendbar.

Lose Kopplung (low coupling) beschreibt das Ziel, möglichst wenig Abhängigkeiten von Elementen untereinander zu haben. Je höher die Abhängigkeiten sind, desto schwerer ist es, das Element wiederzuverwenden. Weiterhin erhöht es die Komplexität des Entwurfs und verringert die Testbarkeit der Klasse.

Eine weitere Unterstützungmöglichkeit beim objektorientierten Entwurf sind Entwurfsmuster. Diese bieten bewährte Lösungsansätze für wiederkehrende Probleme in der Softwareentwicklung. Eine weitergehende Einführung befindet sich im Kapitel Entwurfsmuster.

2.1.3 Aspektorientierung

Aspektorientierung ist ein Paradigma, welches die Objektorientierung erweitert. Ziel ist es, bestimmte Funktionalitäten bei verschiedenen Klassen wiederverwenden zu können (Cross-Cutting Concern). Den Namen verdankt die Aspektorientierung der Tatsache, dass logische Aspekte wie Logging, Transaktionssicherheit und Fehlerbehandlungen von der Geschäftslogik getrennt werden. Dieser Architekturstil bietet die Möglich-

keit, Funktionalitäten nachträglich bzw. sauber getrennt ins das System einzubetten. Ein Beispiel ist die Einführung eines Sicherheitsmodells. Hierbei wird geprüft, ob der aktuelle Benutzer das Recht hat, bestimmte Funktionalität auszuführen. Dieses Vorgehen ermöglicht eine einfache Umsetzung eines komplexen Lizenzmodells oder eines sicheren Plug-in-Systems. Eine weitere Option ist das Einweben eines erweiterten Logging-Aspekts. Sollten während der Entwicklung Laufzeitprobleme entstehen, kann dieser zum Einsatz kommen. Als Beispiel könnte für jede Methode ein Log-Eintrag erstellt werden, welche für ihre Ausführung länger als 500 ms benötigt.
Eine populäre Umsetzung von Aspektorientierung im .NET ist PostSharp[2]. Es basiert darauf, dass man bei den Geschäftslogik-Objekten die Aspekte mittels Attribute angibt:

```
1  internal class BusinessLogic
2  {
3    [NLogAspect]
4    void Init()
5    {
6      // ....
7    }
8  }
```

Listing 2.1: Logging in einer BusinessLogic Klasse über Aspektorientierung.

Die konkrete Umsetzung, was durch das Logging geschehen soll, wird in den Aspekt ausgelagert:

```
1  [Serializable()]
2  public class NLogAspect : OnMethodBoundaryAspect
3  {
4    private static Logger logger = LogManager.GetCurrentClassLogger();
5    public override void OnEntry(MethodExecutionEventArgs eventArgs)
6    {
7      logger.Trace("--- Calling: {0} ---",
8        eventArgs.Method);
9    }
10   public override void OnExit(MethodExecutionEventArgs eventArgs)
11   {
12     logger.Debug("--- Leaving: {0} ---",
13       eventArgs.Method);
14   }
15   public override void OnSuccess(MethodExecutionEventArgs eventArgs)
16   {
17   }
18   public override void OnException(MethodExecutionEventArgs eventArgs)
19   {
```

[2] PostSharp kann über die Internetseite `http://www.sharpcrafters.com/` bezogen werden.

```
20      logger.Fatal("—— Exception: {0} @ {1} ——",
21        eventArgs.Exception.Message, eventArgs.Method);
22    }
23  }
```

Listing 2.2: Einfache Implementierung eines Log-Aspekts.

PostSharp führt eine nachträglichen Kompilierung der Assemblies durch und fügt die zusätzlich benötigten IL-Befehle ein. Abbildung 2.2 zeigt die Funktionsweise von PostSharp.

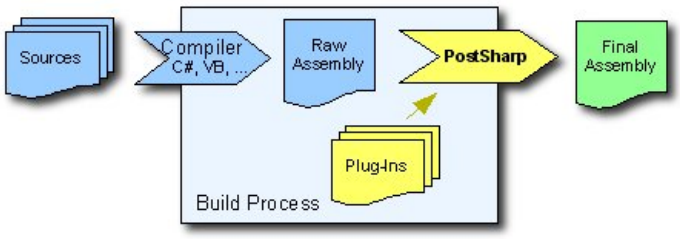

Abbildung 2.2: Funktionsweise von PostSharp (Bildquelle: PostSharp)

Je nach Umsetzung der Aspektorientierung ergeben sich verschiedene Nachteile:

- Längere Kompilierungszeit
- Erschwerte Fehlersuche
- Leicht längere Laufzeiten, durch den nachträglich eingewebten Quelltext

2.1.4 Dienstorientierung

Dienstorientierung (Service-oriented), auch als Serviceorientierung bezeichnet, wird oft in Verbindung mit verteilten Systemen genannt. Bei der Dienstorientierung werden Dienste (Services) beschrieben. Weiterhin wird aufgezeigt wie diese am Besten zu strukturieren und zu nutzen sind. Ein Dienst verfügt über eine klar definierte Schnittstelle und erfüllt eine fest definierte Aufgabe. Auch in der Entwicklung von Desktop-Applikationen spielt die Serviceorientierung eine wichtige Rolle.
Im Vergleich zur Objektorientierung ist bei Serviceorientierung die Funktionalität vom Datenobjekt getrennt. Dies bewirkt einen Performancevorteil, bei einer großen Menge an Objekten. Per Definition sind Services zustandslos, das heißt sie beziehen alle notwendigen Informationen aus Datenobjekten und speichern mögliche Ergebnisse in diesen ab. Dies bietet verschiedene Vorteile:

- Services haben eine hohe Kohäsion, da der Service auf eine Aufgabe bezogen ist

- Die Struktur der Services ermöglicht mit hoher Wahrscheinlichkeit die parallele Verarbeitung verschiedener Datenobjekte
- Services sind dynamisch ladbar und entladbar
- Eine Verkettung von Services mittels "Pipes and Filters"-Prinzip ist möglich. Dies bewirkt zusätzlich eine Vereinfachung und verbesserte Strukturierung des Quelltextes
- Erwähnte Datenobjekte können selbstverständlich als Objekte des objektorientierten Entwurfs angesehen werden

Die Nachteile der Serviceorientierung sind:

- Die Abbildung kompletter Geschäftslogiken als Service ist nicht ohne Weiteres realisierbar
- Insbesondere die Gewährleistung der Datenintegrität ist per Definition nicht gegeben

Ein Beispiel für Serviceorientierung ist das Drucken. Bei einem Druckservice werden Daten aus Objekten ausgelesen, angeordnet und gedruckt. Der Service entscheidet anhand von Daten aus dem Objekt, wie gedruckt werden soll. Es ist nicht notwendig, dass die Logik in jedem Objekt implementiert werden muss. Eine andere Verwendungsmöglichkeit zeigt das Windows Betriebssystem, welches auch eine Reihe von Services bietet die von unterschiedlichen Programmen genutzt werden können (Druckerwarteschlange, Taskplaner).

2.2 Architekturmuster

Dieses Kapitel beschäftigt sich mit wichtigen Ansätzen zur Lösung, wiederkehrender Probleme bei der Softwarearchitektur für komplexe Softwaresysteme.

2.2.1 Domänen-Modelle

In der Fachliteratur werden Domänen-Modelle wie folgt beschrieben [Starke, Seite 41]:

> "Dieses Modell besteht aus fachlich wichtigen Abstraktionen dieser Domäne. Deren Zusammenwirken reflektiert die fachlichen Abläufe und Prozesse."

Domänen-Modelle drücken aber auch zusätzlich das Vokabular und die Kernkonzepte einer Problemdomäne aus. In diesem Modell können verschiedenste Arten von Informationen hinterlegt sein:

- Abhängigkeiten/Verbindungen von Objekten

- Kontrakte die Objekte erfüllen (z.B. implementierte Interfaces, Art und Typ von bestimmten Variablen)
- Regeln (z.B. ein Produkt kann nur mit 7% oder 16% versteuert werden)
- Konkrete Geschäftsregeln (Domänenspezifika), in einem Kassensystem kann folgende Geschäftsregel existieren: Ein Kunde, der innerhalb eines Geschäftsjahres mehr als 10000€ Umsatz generiert hat, wird Premium-Kunde. Dies bedeutet für das nächste Geschäftsjahr, dass ihm 1% Nachlass auf alle Bestellungen gewährt wird.

Erstellt wird das Domänen-Modell von Architekten und Fachexperten der entsprechenden Domäne. Fachexperten sind keine Softwareentwickler, sondern Praktiker der entsprechenden Domäne. Folgende Fachexperten wären denkbar:

- Ärzte in der Medizinsoftware
- Betriebswirte für Finanzsoftware
- Anwälte für Steuersoftware
- Lageristen sowie Logistiker für Lagerverwaltungssysteme

Häufig existiert in großen Firmen eine Fachabteilung mit Kollegen, welche das entsprechende Know-how besitzen. [Starke, Seite 41] schreibt:

> "**Erstellen Sie (möglichst) immer ein Domain Model** Wir halten ein robustes und zwischen Fachseite und Systementwicklung abgestimmtes Domain Model für eine wichtige Grundlage effektiver Systementwicklung. Erstellen Sie grundsätzlich ein solches Modell - es sei denn, Sie arbeiten an völlig trivialen Problemen!"

Abbildung 2.3 zeigt ein einfaches Domain Model für eine Laufsport-Veranstaltung. Dieses Modell müsste schrittweise konkretisiert (das Kapitel Spezialisierung Domänen-Modell zeigt dies) und anschließend implementiert werden.

Es gibt verschiedene Möglichkeiten das Modell zu realisieren. Eine Möglichkeit sind standardisierte Werkzeuge wie die Unified Modelling Language (UML). Durch diese wurde eine Diskussionsgrundlage für Softwarearchitekten und Softwareentwickler geschaffen. Insbesondere die grafisch intuitive Notation führte zum Erfolg von UML. Ein besonderer Vorteil der UML liegt in der Maschinenlesbarkeit. Mittlerweile bietet der Markt eine bereite Platte von UML-Tools[3], welche den Entwicklungsprozess unterstützen. Angefangen bei einer einfachen Möglichkeit, die Entwürfe grafisch zu erstellen, über die Verwaltung verschiedener Entwürfe, bis hin zu einer automatisierten Erzeu-

[3] Computer-Aided Software Engineering-Tools (CASE), bieten die Möglichkeit automatisiert aus fachlichen Beschreibungen Software zu erstellen. Bekannte Vertreter sind Rational Rose, Enterprise Architect und AgroUML.

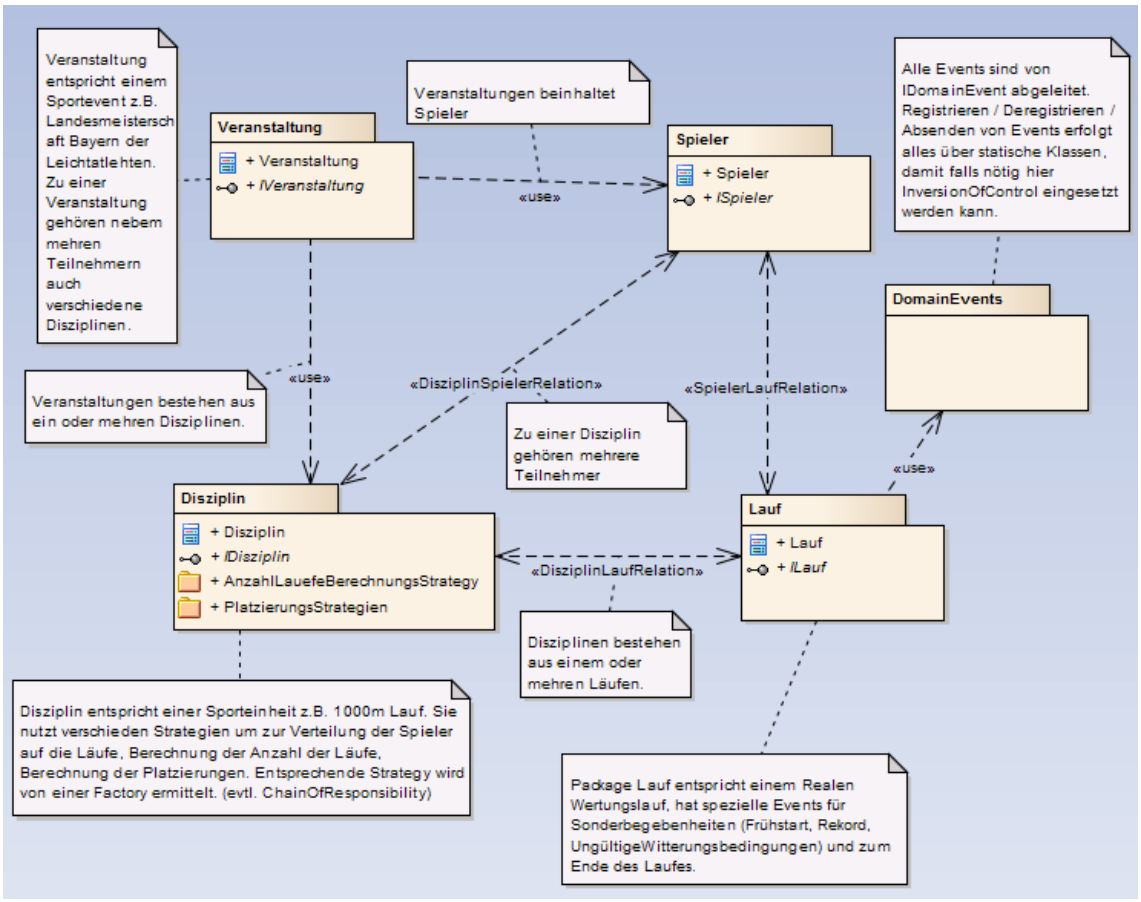

Abbildung 2.3: Domänen-Modell einer Laufsport-Veranstaltung

gung von Quelltext mittels Quelltextgeneratoren[4].
Ob und auf welche Art ein Domain Model benötigt wird, ist eine Entscheidung der Entwicklung. Es empfiehlt sich in hoch komplexen Softwaresystemen auf keines zuverzichten. Den zentralen Satz für die Beschreibung von Domänen-Modellen liefert [Larman, Seite 44]:

> "Beachten Sie, dass ein Domänen-Modell keine Beschreibung von Softwareobjekten darstellt; es ist eine Visualisierung der Konzepte oder der mentalen Modelle eines Gegenstandsbereiches. Deshalb wird das Modell auch als *konzeptuelles (begriffliches) Objektmodell* bezeichnet."

Zusammenfassend hilft das Erstellen eines Domänen Modells dem Softwarearchitekten die Problemdomäne zu erschließen. Das Domänen-Modell wird mit Fachexperten erstellt und bietet eine Diskussionsgrundlage, die unabhängig von der Softwaretechnik ist. Martin Fowler beschreibt den Grund, warum man Domänen Modelle erstellen sollte ([Fowler99, Seite 4]):

[4] Eric Evans hat den Begriff Domain Driven Development geprägt. Es beschreibt eine Methode/Denkweise, wie man Anwendungsdomänen-getrieben das Designs für eine Software erstellt.

> "Einer der Hauptgründe, weshalb ich Analyse- und Entwurfsmethoden verwende, ist die Einbeziehung der Experten des Problembereichs. Dies ist während der konzeptionellen Modellierung essenziell wichtig, denn ich glaube, daß effektive Modelle nur durch Leute erstellt werden können, die den Problembereich genau verstehen. Dies sind Vollzeitmitarbeiter des Problembereichs und nicht Softwareentwickler, auch wenn sich letztere noch so lange mit dem entsprechenden Problembereich beschäftigt haben."

2.2.2 Dynamisch ladbare Funktionalitäten

Die Fähigkeit eines komplexen Softwaresystems, Funktionalitäten zu laden und wieder zu entladen bringt Variabilität in das System. Zum einen werden Ressourcen (z.B. Speicher) erst dann belegt, wenn sie benötigt werden, zum anderen könnten Ressourcenengpässe überstanden werden, indem man Funktionalitäten entlädt. Über die selbe Methodik wie das Laden/Entladen von eigenen Funktionalitäten ist es auch möglich ein Erweiterungssystem (plugin system) zu realisieren. Die .NET CLR bietet dafür Reflektion (Reflection). Reflektion kann Informationen über unbekannte Typen liefern:

- Welche Konstruktoren gibt es
- Die Eigenschaften (Properties), Methoden (Methods), Felder (Fields), welche der Typ beinhaltet
- Sichtbarkeit der einzelnen Elemente
- Welche Meta-Attribute sind der Klasse, den Eigenschaften oder den Methoden zugeordnet
- Die Basisklasse sowie die vom Typ implementierten Schnittstelle (Interface)

Darüber hinaus ist es möglich, Objekte von Typen zu erzeugen und zu verwenden, deren Typendefinition sich in einem nicht referenzierten Assembly befindet. Ein Vorteil von dynamisch ladbaren Funktionalitäten ist es, dass das initiale Starten der Software schneller wird, da die Funktionen erst geladen werden, wenn sie benötigt werden (lazy loading). Als Beispiel für eine dynamisch ladbare Funktionalität dient das Drucken. Es wird erst gebraucht, wenn der Nutzer etwas drucken möchte. Die Quelltextanteile, welche das Drucken realisieren, (inkl. Ressourcen, z.B. Bilder), sollten erst dann geladen werden, wenn Bedarf besteht. Es sind viele Anwendungen denkbar, in denen der Benutzer meist Dokumente lediglich bearbeitet ohne sie zu drucken. Ein weit verwendeter Anwendungsfall ist die Entkopplung von Funktionalität mittels Dependency Injection. Hierfür kommt eine Komponente zur Anwendung, die mittels einer Konfigurationsdatei (beispielsweise XML) die entsprechende Ladefunktionalität implementiert. Objekte, die Funktionalität kapseln, werden nicht mehr direkt an der Verwendungsstelle erzeugt sondern über diese Komponente angefordert. Neben der Trennung von Interface und Implementierung zeigt sich die Mächtigkeit dieses Konzepts dadurch, dass durch Änderung der Konfigurationsdatei Funktionalität ausgetauscht werden kann. So

kann mittels einer geänderten Konfigurationsdatei ein alternativer Druckservice geladen werden, wobei eine weitere Anpassung des bestehenden Quelltextes nicht notwendig ist. Neben der hohen Flexibilität erhöht ein Konzept so die Wiederverwendbarkeit und Testbarkeit.

```
1  //class which should be loaded dynamical (contained in demo.dll)
2  public class Trace
3  {
4    private Queue<string> m_MessageBuffer;
5
6    [MethodImpl(MethodImplOptions.Synchronized)]
7    public Trace()
8    {
9      m_MessageBuffer = new Queue<string>(500);
10   }
11   public void Log(string message, params string[] parameter)
12   {
13     m_MessageBuffer.Enqueue(string.Format(message, parameter));
14   }
15   public void Save(string filePath)
16   {
17     using (TextWriter tw = new StreamWriter(filePath))
18     {
19       while (m_MessageBuffer.Count != 0)
20       {
21         tw.WriteLine(m_MessageBuffer.Dequeue());
22       }
23       tw.Flush();
24       tw.Close();
25     }
26   }
27 }
28
29 //class which load and use Trace (contained in demo.exe)
30 class Program
31 {
32   static void Main(string[] args)
33   {
34     Assembly ass = Assembly.Load(
35       new AssemblyName(
36         "demo, Version=1.0.0.0, Culture=null, PublicKeyToken=null"
37         )
38       );
39     Type t = ass.GetType("demo.Trace");
40
41     object obj = Activator.CreateInstance(t);
```

```
42      t.InvokeMember("Log",
43        BindingFlags.InvokeMethod,
44        null,
45        obj,
46        new[] { "Hallo {0}", "Welt" });
47      t.InvokeMember("Save",
48        BindingFlags.InvokeMethod,
49        null,
50        obj,
51        new[] { @"C:\demo.log" });
52    }
53  }
```

Listing 2.3: Über Reflektion realisiertes Tracing.

Da der hierfür notwenige Quelltext von erhöhter Komplexität ist, ist es ratsam, eine entsprechende Zugriffsschicht zu erstellen. Dieser vielseitige Ansatz wird in verschiedenen Anwendungsfällen verwendet. Beispielsweise basieren Aktuelle .NET Oberflächentechnologien,wie z.B. WPF, Silverlight und METRO, auf diesem Konzept.
Trotz aller Flexibilität bringt dieser Ansatz auch Nachteile mit sich:

- Konfigurationsfehler treten erst zur Laufzeit auf
- Das Laufzeitverhalten verschlechtert sich und es kommt zu einem erhöhten Speicherverbrauch
- Das Entladen von Assemblies ist nicht ohne Weiteres möglich
- Sicherheitsbedenken (beispielsweise bei einem via Reflektion realisiertem Erweiterungssystem)

In der Praxis bietet die CLR eine Handhabe - die Applikationsdomänen (App domains). Eine AppDomain ist ein Behälter (Container) für ein oder mehrere Assemblies. Ihre einzige Aufgabe besteht darin, eine Isolation zu gewährleisten. Bei jeder Initialisierung der CLR wird eine Standard-AppDomain (Default AppDomain) erzeugt. Dies geschieht beispielsweise beim Starten eines .NET Programmes. In der Standard-AppDomain werden alle Assemblies des Programmes geladen. Diese wird erst zerstört, wenn das Programm geschlossen wird. Um dies zu umgehen ist es möglich, selbst AppDomains anzulegen, die jederzeit entladen werden können. Entladen bedeutet hierbei, dass Assemblies aus dem Speicher entfernt werden und alle der AppDomain zugehörigen Objekte freigegeben werden. Freigegeben bedeutet, dass die Objekte auf dem Heap markiert werden. Ein expliziter Aufruf von GC.Collect() ist also nicht notwendig. Das folgende Listing zeigt die veränderte Main Methode (aus dem oberen Beispiel-Quelltext), sodass eine eigene AppDomain verwendet wird:

```
1  class Programm
2  {
3    static void Main(string[] args)
```

```
4   {
5     AppDomain ap = AppDomain.CreateDomain("myAppDomain");
6     ObjectHandle oh = ap.CreateInstance(
7       "demo, Version=1.0.0.0, Cluture=null, PublicKeyToken=null",
8       "demo.Trace");
9     object obj = oh.Unwrap();
10    Type t = obj.GetType();
11    t.InvokeMember("Log",
12      BindingFlags.InvokeMethod,
13      null,
14      obj,
15      new[] { "Hallo {0}", "Welt" });
16    t.InvokeMember("Save",
17      BindingFlags.InvokeMethod,
18      null,
19      obj,
20      new[] { @"C:\demo.log" });
21    AppDomain.Unload(ap);
22  }
23 }
```

Listing 2.4: Reflektion in Verbindung mit AppDomains.

2.2.3 Model-View-ViewModel

Model-View-ViewModel (MVVM) ist ein Architekturmuster zum Aufbau von Oberflächen in WPF oder Silverlight. Es wurde vom Microsoft Patterns and Practices Team als Spezialisierung des von Martin Fowler eingeführten *Presentation Model* entwickelt. Das Pattern und Practices Team hat unter dem Namen Prism[5] eine Anleitung zur Verwendung von MVVM bereitgestellt.
MVVM basiert darauf, dass die View (z.B. das Oberflächensteuerelement sowohl XAML Anteil als auch CodeBehind) nicht über einen Controller auf dem Modell arbeitet wie es beim bekannten MVC Pattern (Model-View-Controller) der Fall ist, sondern die View arbeitet nur gegen ein Oberflächenmodell. Das Oberflächenmodell enthält das Wissen, wie es sich auf das konkrete Modell abbildet.
Die View erhält das ViewModel, indem das ViewModel der Eigenschaft *DataContext* zugewiesen wird. Die Interaktion zwischen View und ViewModel erfolgt vollkommen lose. Gekoppelt über Namen (Binding-By-Name) von *Eigenschaften, überwachbare Aufzählungen (observable collections) und Kommandos (Commands)*, welche das ViewModel bereit stellt. In dem folgenden Listing bekommt ein TreeView seine anzuzeigenden Items über die Eigenschaft "FirstLevelElements" von dem ViewModel. Ob es Unterelemente gibt, erfährt die View beim Auslesen der Eigenschaft "Elements" eines ViewModel-

[5] Prism ist Verfügbar unter `http://msdn.microsoft.com/en-us/library/gg406140.aspx`

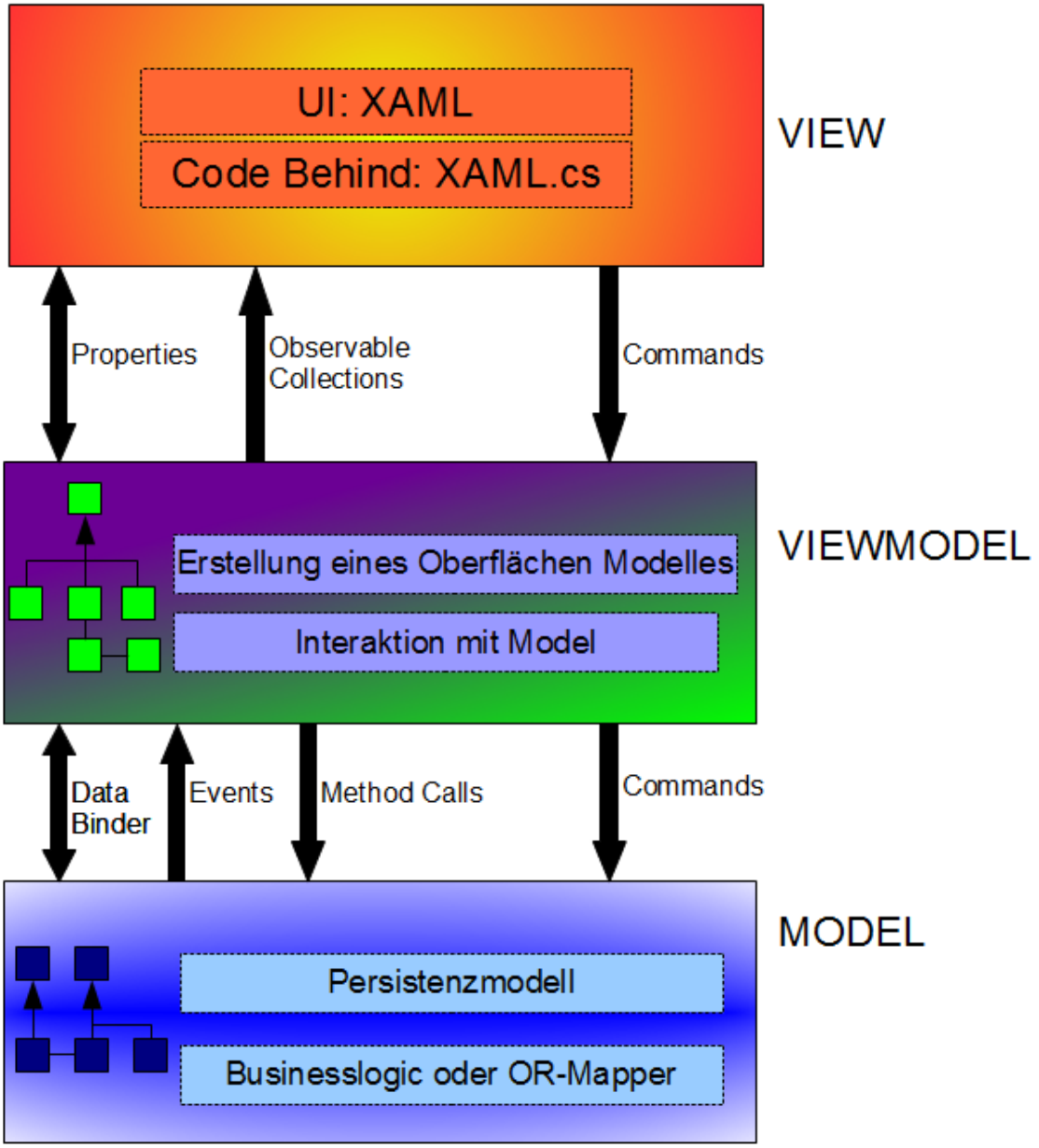

Abbildung 2.4: Überblick des Aufbaus des Model-View-ViewModels

Elementes. Alle ViewModel-Elemente bieten zusätzlich die Eigenschaften "IsSelected" und "IsExpanded" an, welche auf die gleichnamigen Eigenschaften eines TreeView-Items gebunden wird.

```
<TreeView ItemsSource="{Binding FirstLevelElements}">
  <TreeView.ItemContainerStyle>
    <Style TargetType="{x:Type TreeViewItem}">
      <Setter
        Property="IsExpanded"
        Value="{Binding IsExpanded, Mode=TwoWay}" />
      <Setter
```

```
            Property="IsSelected"
            Value="{Binding IsSelected, Mode=TwoWay}" />
      </Style>
    </TreeView.ItemContainerStyle>
    <TreeView.ItemTemplate>
      <HierarchicalDataTemplate ItemsSource="{Binding Elements}">
        <TextBlock Text="{Binding Text}" />
      </HierarchicalDataTemplate>
    </TreeView.ItemTemplate>
</TreeView>
```

Listing 2.5: Anbindung der View an das ViewModel über Properties

Das Schlüsselwort "Mode=TwoWay" bedeutet hierbei, dass die Eigenschaft des ViewModels nicht nur von der View gelesen und geschrieben wird, sondern auch vom ViewModel. Die Standard Schnittstelle *INotifyPropertyChanged* beinhaltet ein Event, dieses überwacht die View automatisch. Dieser Standardmechanismus führt zu einer automatischen Aktualisierung der View, sollte sich eine Eigenschaft mit entsprechenden Namen verändert. Das setzt voraus, dass das ViewModel bei einer Änderung ein entsprechendes Event auslöst. Der folgende Quelltext zeigt eine beispielhafte Implementierung:

```
1  public class ViewModelElement : INotifyPropertyChanged
2  {
3    private bool m_IsExpanded;
4    public bool ThrowOnInvalidConfiguration { get; set; }
5    public bool IsExpanded
6    {
7      get { return m_IsExpanded; }
8      set
9      {
10       if (value != m_IsExpanded)
11       {
12         m_IsExpanded = value;
13         this.RisePropertyChanged("IsExpanded");
14       }
15     }
16   }
17   private void RisePropertyChanged(string name)
18   {
19     VerifyPropertyName(name);
20     if (PropertyChanged != null)
21     {
22       PropertyChanged.Invoke(this,
23         new PropertyChangedEventArgs(name));
24     }
25   }
```

```
26    [Conditional("DEBUG")]
27    [DebuggerStepThrough]
28    private void VerifyPropertyName(string propertyName)
29    {
30      if (TypeDescriptor.GetProperties(this)[propertyName] == null)
31      {
32        const string m_ErrorMessage = "Invalid property name: {0}";
33        if (ThrowOnInvalidConfiguration)
34        {
35          throw new Exception(
36            string.Format(m_ErrorMessage, propertyName));
37        }
38        else
39        {
40          Debug.Fail(string.Format(m_ErrorMessage,
41          propertyName));
42        }
43      }
44    }
45  }
```

Listing 2.6: Informierung der View über Änderungen durch das ViewModel mit INotifyPropertyChange

Um Aufgaben von der View an das ViewModel zu deligieren, werden Commands verwendet.

```
<Button
  Content="Find"
  Command="{Binding␣ViewModelSearchingCommand}" />
```

Listing 2.7: Ansteuerung der Funktionalitäten des ViewModels durch die View über Commands

Das ViewModel muss eine Klasse, welche durch die Schnittstelle "ICommand" implementiert wird, über ein Property (hier mit dem Namen ViewModelSearchingCommand) zugreifbar machen. Innerhalb des Commands kann das ViewModel durch Setzen von Eigenschaften innerhalb des ViewModels das Ergebnis an die Oberfläche zurückmelden.
Das führt zu einer sehr losen Kopplung und hat zwei Vorteile:

1. Es ist möglich die View gegen eine andere zu ersetzen, welche dieselben Properties, Commands und Observable Collections verwendet (Interfacestabilität). MVVM kann als spezielles Schichtenmodell für Oberflächenentwicklung gesehen werden.
2. Die Logik, welche auf dem Model arbeitet, ist im ViewModel gekapselt. Dies ermöglicht das implementieren vollständig automatischer Tests.

Das ViewModel ist dahingehend fester mit dem Model verbunden. Es hat Kenntnis über den Aufbau des Models und wie die Abbildung auf die eigenen Elemente des ViewModels zu erfolgen hat. Zur Interaktion mit dem Model stehen viele Möglichkeiten zur Verfügung:

- Data-Binder (eignen sich für eine Eins-zu-Eins-Abbildung)
- Commands
- Direkter Methodenaufruf über Schnittstellen
- Properties
- Events (zum Beispiel zur Aktualisierung bei Änderungen am Model)

Notwendig ist hierbei die vollständige Unabhängigkeit des Models von dem ViewModel. Bei einer nach MVVM erstellten Oberfläche ist es möglich, möglich die Oberfläche ohne Änderung des Models auszutauschen, auch ist das Model unabhängig testbar. Sollten darüber hinaus auch die Schnittstellen zum Model klar definiert sein, können auch Teile des ViewModels automatisiert getestet werden. Das geschieht beispielsweise durch Implementierung der Model Schnittstellen durch Stellvertreter (Mocked Objects)[6]. Abbildung 2.5 zeigt eine Anwendung, die mit MVVM realisiert wurde. Der Quelltext befindet

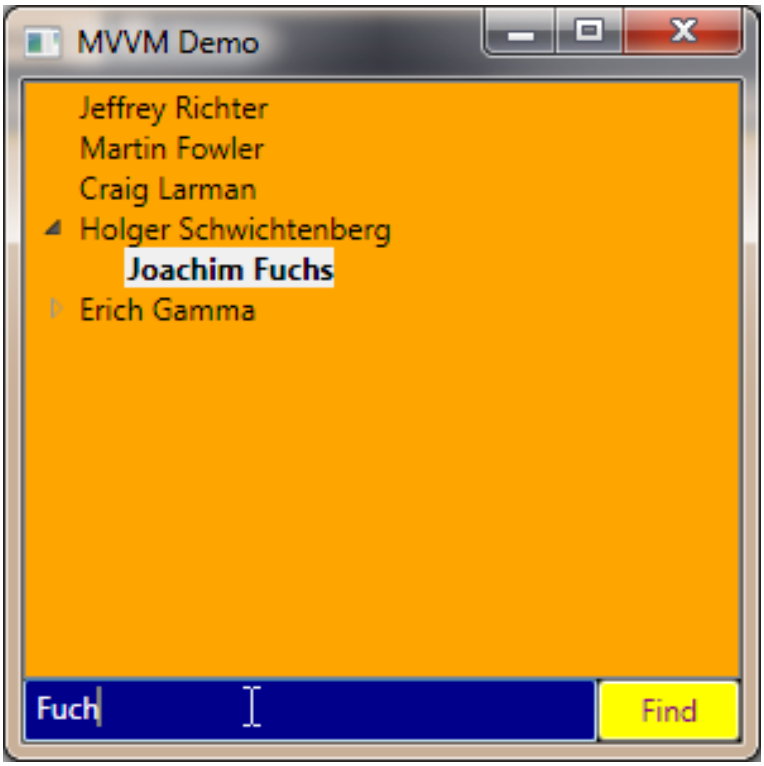

Abbildung 2.5: WPF Anwendung mit Suchfunktion in einer Aufzählung von Autoren

[6] Mock Objects simulieren reale Objekte. Sie helfen damit andere Objekte, welche die gemockten Objekte verwenden, seperat zu testen. Dies ist sehr hilfreich, wenn z.B. die realen Objekte zum Funktionieren eine Datenbankverbindung benötigen.

sich im Kapitel Anwendung entworfen nach MVVM.
Neben den Vorteilen der Test- und Austauschbarkeit gibt es folgende Nachteile:

- In aufwendigen Views, im Sinne eines ganzen Fensters mit Abhängigkeiten zwischen den Oberflächensteuerelementen, sind auch die Abhängigkeiten der ViewModel-Elemente ähnlich. Daher entsteht fast identischer Quelltext.
- Für einfache Oberflächen ist Model-View-ViewModel zu aufwendig zu implementieren.
- Für aufwendige Oberflächen ist es schwer das ViewModel von vornherein so zu entwerfen, dass es ein großes Maß an Allgemeingültigkeit bietet. [7]
- Deklarative Bindung über Namen erschwert die Fehlersuche. Es ist nicht immer möglich Haltepunkte im Quelltext zu setzen, da klassische Entwicklungswerkzeuge hierbei keine Unterstützung bieten.
- Durch den dynamisch deklarativen Ansatz ist es anspruchsvoll, komplexere Views zu realisieren. Insbesondere falls dieselben Informationen an verschiedenen Stellen der View bearbeitet werden sollen.

2.2.4 Verwalteter und Nativer Quelltext

Allgemein Interoperabilität (Interop) zwischen verwaltetem und nativem Quelltext ist ein großes Thema für komplexe Softwaresysteme. Sei es alte Funktionalität (Legacy Code) die man nicht neu schreiben will/kann, der Aufruf einer WinAPI-Funktion, welche noch keine Entsprechung in der .NET Klassenbibliothek hat oder einfach dazu gekaufte Komponenten. Der Zugriff von nativem Quelltext auf verwaltetem ist oft unumgänglich.

Verwaltet ⇒ Nativ Dies ist für eine .NET-Anwendung der Hauptanwendungsfall. Es gibt verschiedene Wege um aus .NET auf native Funktionalitäten zuzugreifen. Der "normale" Weg ist das Arbeiten mit P/Invoke (Platform Invoke). Der Folgende Quelltext nutzt P/Invoke um eine MessageBox mittels WinAPI-Funktionen anzuzeigen.

```
1 class Program
2 {
3   static void Main(string[] args)
4   {
5     Notify.Foo(IntPtr.Zero,
6       "Hallo Welt!",          //text
7       "The program says",     //caption
8       0);                     //only okay button
9   }
```

[7] Diese Aussage ist eine Übersetzung aus dem Englischen. Das Original ist von John Gossmann aus seinem Blog `http://blogs.msdn.com/b/johngossman/archive/2006/03/04/543695.aspx`

```
}
class Notify
{
  [DllImport("user32.dll",
    CharSet = CharSet.Auto,
    EntryPoint = "MessageBox")]
  public static extern uint Foo(
    IntPtr hWnd,      //parent
    String text,      //description
    String caption,   //form text
    int options);     //which buttons should shown
}
```

Listing 2.8: Verwendung einer WinAPI Funktion über P/Invoke.

Eine zweite Variante zum Ausführen von nativem Maschinencode ist es diesen via COM anzusprechen. Für COM-Assemblies ist eine COM-Typdefinitionsdatei (*.tlb) vorhanden. Das Werkzeug Tlbimp.exe[8] generiert aus dieser Typdatei eine CLR-Entsprechung. Damit ist der Zugriff auf COM-Objekte ohne weiteres möglich.

Nachfolgend ist der Inhalt einer InterfaceDescriptionLanguage-Datei (*.idl) zu sehen, aus welcher mittels eines Compilers die korresspondierende Typdefinitionsdatei generiert wird:

```
[
  uuid (10000005-0000-0000-0000-000000000000),
  helpstring ("GetComputerName 1.0 Type Library"),
  version (1.0)
]
library GetComputerName
{
  importlib ("stdole.tlb");

  interface _IGetComputerNameInterface;
  [
    uuid (10000011-0000-0000-0000-000000000000),
    object,
    helpstring("IGetComputerNameInterface Interfaces")
  ]
  interface _IGetComputerNameInterface : IUnknown
  {
    import "unknwn.idl";
    HRESULT GetComputerName([out] BSTR* bName);
  }
```

[8] Teil des .NET Framework SDKs. Weitere Information unter `http://msdn.microsoft.com/de-de/library/tt0cf3sx%28v=vs.80%29.aspx`

```
21
22    [
23      uuid (10000042-0000-0000-0000-000000000000),
24    ]
25    coclass IGetComputerNameInterface {
26      [default] interface _IGetComputerNameInterface;
27    };
28  }
```

Listing 2.9: Interface-Beschreibung einer COM-Schnittstelle zur Ermittlung des Computernamens.

Mit der generierten Typdefinitionsdatei erzeugt Tlbimp.exe eine C# Library mit folgendem Inhalt[9].

```
1  using System;
2  using System.Runtime.CompilerServices;
3  using System.Runtime.InteropServices;
4  namespace GetCompNameLib
5  {
6    [InterfaceType(1), Guid("10000011-0000-0000-0000-000000000000")]
7    public interface _IGetComputerNameInterface
8    {
9      [MethodImpl(MethodImplOptions.InternalCall)]
10     void GetComputerName([MarshalAs(UnmanagedType.BStr)] out string
           bName);
11   }
12   [Guid("10000011-0000-0000-0000-000000000000"),
13   CoClass(typeof(IGetComputerNameInterfaceClass))]
14   public interface IGetComputerNameInterface :
15   _IGetComputerNameInterface
16   {
17   }
18
19   [TypeLibType(2),
20   Guid("10000042-0000-0000-0000-000000000000"),
21   ClassInterface(0)]
22   public class IGetComputerNameInterfaceClass :
23   _IGetComputerNameInterface,
24   IGetComputerNameInterface
25   {
26     [MethodImpl(MethodImplOptions.InternalCall)]
27     public extern IGetComputerNameInterfaceClass();
28     [MethodImpl(MethodImplOptions.InternalCall)]
```

[9] Für das Reverse Engineering von .NET Assemblies eignen sich der kostenpflichtige .NET Reflector (`http://www.reflector.NET/`) oder der (open source) ILSpy (`http://wiki.sharpdevelop.NET/ILSpy.ashx`).

```
29      void _IGetComputerNameInterface.GetComputerName
30      ([MarshalAs(UnmanagedType.BStr)] out string bName);
31    }
32  }
```

Listing 2.10: Aus Typdatei mittels tlbimp.exe erzeugter C# Zugriffscode.

Um die COM-Klasse im Programm zu benutzen, muss nur die C# Library referenziert werden. Die Verwendung ist sehr einfach:

```
1  using System;
2  using GetCompNameLib;
3
4  class Program
5  {
6    static void Main(string[] args)
7    {
8      string computerName;
9      IGetComputerNameInterfaceClass test =
10       new IGetComputerNameInterfaceClass();
11     test.GetComputerName(out computerName);
12
13     Console.WriteLine(computerName);
14   }
15 }
```

Listing 2.11: Verwendung der von tlbimp.exe erzeugten C# Zugriffsklassen.

Nativ $\Leftarrow$ Verwaltet Aus einer nativen Anwendung ist die Verwendung von .NET Assemblies möglich. Dazu gibt es im wesentlichen drei Varianten. Die erste Variante, - Zugriff über COM - erfordert Anpassungen im verwaltetem Quelltext.

- Das Assembly-Attribute ComVisible muss auf "true" gesetzt werden
- Schnittstellen-/Klassendefinitionen müssen Guid, COM-Typ und Sichtbarkeit am COM-Sichtbarkeit über Attribute gesetzt werden
- Bei Parametern oder Rückgabewerten von Methoden oder Eigenschaften muss über MarshalAs-Attribut der COM-Typ bestimmt werden

Damit kann über die Werkzeuge Tlbexp.exe[10] oder Regasm.exe[11] die COM Typdatei extrahiert werden.
Die zweite Variante besteht aus der Kompilierung des nativen Quelltext mittels C++/CLI. Dadurch entsteht ein normales .NET Assembly welches einfach in eine verwalteten Anwendung eingebunden werden kann.
Die dritte Variante wird CLR-Hosting genannt. Dazu wird eine Instanz der CLR in den Prozessraum des nativen Prozesses geladen. Dieser Prozess hat folgende Möglichkeiten, um mit der CLR zu arbeiten:

- Initialisieren, Starten und Anhalten
- Assemblys laden und IL-Code ausführen
- Hinterlegen von Callbacks bei Ereignissen (Stacküberlaufausnahme, (ent)laden einer AppDomain, Beenden der CLR)
- Verwendung von bestimmten Klassen/Member verbieten
- Einflussnahme auf Entscheidungen der CLR (Speicherreservierung, Threadsynchronisierung, Laden von Assemblys und andere)
- Notifikationen bei Beginn oder Ende einer GarbageCollection

Die Vorteile des CLR-Hosting beschreibt Jeffrey Richter so (aus [Richter, Seite 535]):

- "Für die Programmierung kann eine beliebige Programmiersprache benutzt werden.
- Code wird vom JIT-Compiler auf Geschwindigkeit optimiert (und nicht von einem Interpreter ausgeführt).
- Dank Garbage Collection vermeidet der Code Speicherlecks und Speicherbeschädigung.
- Code läuft in einer sicheren Sandbox.
- Der Host braucht sich nicht darum zu kümmern, eine leistungsfähige Entwicklungsumgebung bereitzustellen. Der Host kann vorhandene Technologien nutzen: Sprachen, Compiler, Editoren, Debugger, Profiler und vieles mehr."

Der Nachteil vom CLR-Hosting besteht darin, dass, einmal in einen Windows-Prozess geladen, die CLR nicht entladen werden kann. Das folgende (verwaltete) Beispiel gibt die lokale IP-Adresse zurück:

```
1 namespace CLR_Hosting_Sample
2 {
3     public sealed class Sample
```

[10] Das Type Library Exporter-Tool, ist Teil des .NET Framework SDKs. Es exportiert aus einem .NET Assembly die Typen Datei. http://msdn.microsoft.com/de-de/library/hfzzah2c%28v=vs.80%29.aspx

[11] Das Assembly Registration-Tool, ist Teil des .NET Framework SDKs. Es generiert aus einem .NET Assembly eine Typen Datei und registriert diese als COM-Server im Betriebssystem. http://msdn.microsoft.com/de-de/library/tzat5yw6%28v=vs.80%29.aspx

```
4     {
5         public static int GetLocalIP(string param)
6         {
7             var hostEntry = Dns.GetHostEntry(Dns.GetHostName());
8             var ip = ( from addr in hostEntry.AddressList
9                        where addr.AddressFamily == AddressFamily.
                           InterNetwork
10                       select addr
11                   ).FirstOrDefault();
12            return (int)ip.Address;
13        }
14    }
15 }
```

Listing 2.12: Einfache C# Klasse, welche die loakle IP-Adresse des Rechners zurück gibt.

Die Verwendung der Methode aus einem nativen Prozess sieht wie folgt aus (die verwaltete DLL trägt den Namen CLR_HOSTING_SAMPLE.dll und befindet sich im selben Verzeichnis wie die native Anwendung):

```
1  #include <Windows.h>
2  #include <mscoree.h>
3  #include <iostream>
4  #include <assert.h>
5  #include <metahost.h>
6  #pragma comment(lib, "mscoree.lib")
7
8  void main ()
9  {
10     //loading and init required CLR
11     ICLRMetaHost *pMetaHost = NULL;
12     HRESULT hr = CLRCreateInstance(CLSID_CLRMetaHost,
13                      IID_ICLRMetaHost,
14                      (LPVOID*)&pMetaHost);
15     assert(hr == S_OK);
16
17     WCHAR wszVersion[MAX_PATH] = { 0 };
18     DWORD dwBufferLen = MAX_PATH;
19     hr = pMetaHost->GetVersionFromFile(
20         L"CLR_HOSTING_SAMPLE.dll",
21         wszVersion,
22         &dwBufferLen);
23     assert(hr == S_OK);
24
25     ICLRRuntimeInfo *info;
26     hr = pMetaHost->GetRuntime(wszVersion,
```

```
27              IID_PPV_ARGS(&info));
28      assert(hr == S_OK);
29
30      ICLRRuntimeHost *ptr;
31      hr = info->GetInterface(CLSID_CLRRuntimeHost,
32                      IID_PPV_ARGS(&ptr));
33      assert(hr == S_OK);
34
35      //starting the CLR
36      hr = ptr->Start();
37      assert(hr == S_OK);
38
39      //calling managed implementation
40      DWORD ip;
41      hr = ptr->ExecuteInDefaultAppDomain(
42          L"CLR_HOSTING_SAMPLE.dll",
43          L"CLR_Hosting_Sample.Sample",
44          L"GetLocalIP",
45          NULL,
46          &ip);
47      assert(hr == S_OK);
48
49      //using output ..
50      int part1 = ip & 0xFF000000;
51      int part2 = ip & 0x00FF0000;
52      int part3 = ip & 0x0000FF00;
53      int part4 = ip & 0x000000FF;
54      std::cout << "Local IP-Adress is: "
55          << part4 << "."
56          << (part3 >> 8 ) << "."
57          << (part2 >> 16) << "."
58          << (part1  >> 24 )
59          << std::endl;
60  }
```

Listing 2.13: Hosting der CLR aus C++, zum Auslesen der lokalen IP-Adresse.

2.3 Entwurfsmuster

Dieses Kapitel behandelt die Definition von Entwurfsmustern (Pattern) und soll die wichtigsten nennen. Dies dient der Erschaffung eines einheitlichen Vokabulars für Softwarearchitekten. Die Erklärung einiger oder sogar aller Entwurfsmuster würde den Rahmen dieser Arbeit sprengen. Weiterführende Informationen über Entwurfsmuster bieten unter anderem Online-Pattern-Kataloge. Für Pattern Umsetzung im .NET-Umfeld empfiehlt sich: `http://www.dofactory.com/Patterns/Patterns.aspx`.

2.3.1 Gang of Four Pattern

Als die Gang of Four (GoF) bezeichnet man Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. Sie haben im Oktober 1994 *das Buch* der objektorientierten Software Entwicklung veröffentlicht. Es beschreibt 23 Entwurfsmuster (design patterns, kurz pattern). Entwurfsmuster beschreiben stetig wiederkehrende Probleme in der Softwareentwicklung und deren Lösungsansatz. Es kann nur ein Lösungsansatz sein, da Softwaresysteme sehr unterschiedlich sind. Eine allgemeingültige Lösung kann es daher nicht geben, dafür aber eine Lösungsstrategie, welche auf das Problemfeld anwendbar ist. Es gibt eine im GoF-Buch beschriebene Notation von Patterns. Diese bestehen im Wesentlichen aus den folgenden vier Abschnitten:

1. **Mustername** - dient als Vokabel um diesen Lösungsansatz zu identifizieren.
2. **Problemabschnitt** - beschreibt das Umfeld des Problems und das Problem an sich (besonders wann es anzuwenden ist).
3. **Lösungsabschnitt** - hier steht keine eigentliche Lösung sondern vielmehr eine Schablone, die es auf das konkrete Problem anzuwenden gilt, um es zu lösen. Die Schablone besteht aus Elementen sowie den Beziehungen, Interaktionen und Zuständigkeiten untereinander.
4. **Konsequenzenabschnitt** - beschreibt die Folgen durch den Einsatz des Entwurfsmusters durch Auflistung von Vor- und Nachteilen.

Weitere Definition "Was ist ein Pattern?" (aus [Gamma, Seite 4]):

> "Die Bestimmung dessen, was ein Muster ist und was nicht, hängt von der jeweiligen Perspektive ab. Was aus einer Perspektive als Muster erscheint, stellt aus einer anderen Perspektive betrachtet einen primitiven Baustein dar. In diesem Buch haben wir uns auf eine bestimmte Abstraktionsebene konzentriert. *Entwurfsmuster* befassen sich nicht mit Entwürfen, wie verkettet Listen oder Hash-Tabellen, die als einzelne Klassen programmiert und wiederverwendet werden können. Sie stellen auch keine komplexen für einen Anwendungsbereich spezifische Entwürfe dar, die eine ganze Anwendung oder ein Subsystem realisieren. Die Entwurfsmuster in diesem Buch sind *Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen*.
> Ein Entwurfsmuster benennt, abstrahiert und identifiziert die relevanten Aspekte einer allgemeinen Entwurfsstruktur. Diese Aspekte beschreiben, warum das Muster für die Entwicklung eines wiederverwendbaren objektorientierten Entwurfs nützlich ist. Das Entwurfsmuster identifiziert die teilnehmenden Klassen und Objekte, die Rollen, die sie spielen, die Interaktionen zwischen den Rollen und die ihnen zugeteilten Aufgaben. Jedes Entwurfsmuster konzentriert sich auf ein bestimmtes objektorientiertes Entwurfsproblem. Es be-

> schreibt, wann es einsetzbar ist, ob es angesichts einschränkender Randbedingungen eingesetzt werden kann, und welche Konsequenzen sein Einsatz hat."

GoF-Pattern sind aus der objektorientierten Architekturen für komplexe Softwaresysteme nicht wegzudenken. Den Wert der Entwurfsmuster erfasst man erst, wenn man entweder ein schwieriges Problem gelöst hat, das selbe Problem durch Entwurfsmuster bereits gelöst wurde, oder wenn Pattern den Einbau neuer Funktionalitäten unterstützt hätten (z.B. Erweiterung des Command-Patterns mit Undo- und Redo-Funktionalitäten). Die Entwurfsmuster teilen sich in drei Gruppen auf (Erzeugungsmuster, Strukturmuster und Verhaltensmuster).
Es gibt 5 Erzeugungsmuster:

- Abstrakte Fabrik (abstract factory)
- Erbauer (builder)
- Fabrikmethode (factory method)
- Prototyp (prototype)
- Singleton (singleton)

Es gibt 7 Strukturmuster:

- Adapter (adapter)
- Brücke (bridge)
- Dekorierer (decorator)
- Fassade (facade)
- Fliegengewicht (flyweight)
- Kompositum (composite)
- Stellvertreter (proxy)

Es gibt 11 Verhaltensmuster:

- Befehl (command)
- Beobachter (observer)
- Besucher (visitor)
- Interpreter (interpreter)
- Iterator (iterator)
- Memento (memento)
- Schablonenmethode (template method)
- Strategie (strategy)
- Vermittler (mediator)
- Zustand (state)
- Zuständigkeitskette (chain of responsibility)

GoF-Pattern korrelieren mit den Regeln für objektorientierten Entwurf (GRASP). Zum Beispiel besagt GRASP *Polymorphismus*, dass die Polymorphismus-Eigenschaften von Objekten genutzt werden können, um Fallunterscheidungen zu vermeiden. Genau dies macht das Strategie-Entwurfsmuster.
Eine andere Überlegung, welche hilfreich ist um den "Kern" eines Patterns zu ergründen, ist **Responsibility-Driven Design (RDD)**. Entwickelt wurde RDD von Rebecca Wirfs-Brock und Brian Wilkerson. Es basiert darauf, einen Vertrag (contract) zwischen Softwareobjekten mittels Fragen auszuhandeln:

- Für welche Aktionen ist ein Objekt verantwortlich?
- Welche Informationen teilt ein Objekt mit anderen?

Es klingt trivial, doch durch diesen "Blickwinkel" lassen sich Verantwortlichkeiten besser verstehen. Negativbeispiel: Eine Klasse A übernimmt die Berechnung von Steuersätzen. Dafür benötigt es ein Objekt, welches entweder vom Typ Klasse B oder Klasse C ist. Je nachdem welchen Typ das Objekt hat, wird Klasse A unterschiedliche Berechnungen starten. Warum sind Klasse B und Klasse C nicht selbst für die Berechnung verantwortlich? Wenn sie die Berechnung selbstständig regeln, müssen sie dann weiter Informationen teilen oder ist dies überflüssig? Klasse A soll einfach an dem Objekt die Berechnung starten und mit dem Ergebnis weiterarbeiten, und nicht Wissen über andere Objekte beinhalten.

2.4 .NET Framework

In diesem Kapitel werden theoretische Grundlagen über das .NET Framework und die, sich eventuell auf die Softwarearchitektur auswirkende Eigenschaften vermittelt.

2.4.1 Speicherverwaltung

Da die Common Language Runtime (im folgenden CLR genannt) eine automatische Speicherverwaltung (Memory Management) beinhaltet, braucht und kann der Entwickler das Allokieren und Freigeben von Speicher nicht mehr beeinflussen. Jeffrey Richter schreibt zum Arbeiten auf einer Plattform mit automatischer Speicherverwaltung (aus [Richter, Seite 471]):

> "Jedes Programm benutzt Ressourcen irgendeiner Art, zum Beispiel Dateien, Speicherpuffer, Bildschirmbereiche, Netzwerkverbindungen, Datenbankressourcen und so weiter. In einer objektorientierten Umgebung stellt sogar jeder einzelne Typ eine Ressource dar, die ein Programm benutzen kann. Soll eine dieser Ressourcen verwendet werden, muss Speicher für den Typ angefordert werden. Folgende Schritte sind notwendig, wenn sie

auf eine Ressource zugreifen wollen:

1. Reservieren Sie den Speicher für den Typ mit dem IL-Befehl *newobj*. Der Compiler generiert diesen Befehl, wenn sie in C# den Operator *new* aufrufen.
2. Initialisieren Sie den Speicher, um den Anfangszustand der Ressource herzustellen und die Ressource für den Einsatz vorzubereiten. Der Instanzkonstruktor des Types ist dafür verantwortlich, diesen Anfangszustand herzustellen.
3. Benutzen Sie die Ressource, indem Sie auf die Typmember zugreifen (bei Bedarf mehrmals).
4. Beseitigen Sie den Zustand einer Ressource, um aufzuräumen [...].
5. Geben Sie den Speicher frei. Dies ist alleinige Aufgabe des Garbage Collectors.

Dieser scheinbar simple Vorgang war eine der Hauptursachen für Programmierfehler. Wie oft haben Programmierer vergessen, Speicher freizugeben, nachdem er nicht mehr benötigt wurde? Wie oft haben Programmierer versucht, auf Speicher zuzugreifen, nachdem er bereits freigegeben war?
Bei der unverwalteten Programmierung sind diese beiden Bugkategorien schlimmer als die meisten anderen, weil sie normalerweise nicht vorhersagen können, was dabei passieren wird oder wann es passiert. Wenn sich ihre Anwendung wegen anderer Bugs nicht korrekt verhält, korrigieren sie das Problem. Aber diese beiden Bugkategorien verursachen Ressourcenlecks (das heißt Speicherverbrauch) und die Beschädigung von Objekten (das heißt Instabilität). Das Verhalten der Anwendung wird unberechenbar."

Automatische Speicherverwaltung bedeutet aber *nicht*, dass man sich um Speicherverwaltung keine Gedanken machen braucht. Sowohl für den Architekten als auch dem Programmierer ist Wissen über Funktion, Aufbau und Arbeitsweise notwendig, da sonst leicht Fehler begangen werden die die Performance des Programmes beeinträchtigen. Jeder Thread hat in Win32- und .NET-Anwendungen standardmäßig einen 1 MB großen **Stack (Stapelspeicher)**[12]. Erst seit dem .NET Framework 4.0 ist es beim Anlegen neuer Threads möglich, deren Stackgröße festzulegen[13]. Auf dem Stack befinden sich lokale Variablen und Wertetypen (value types). Wertetypen sind alle von der Klasse System.ValueType abgeleitet. Zu ihnen gehören sowohl einfache Datentypen wie System.Boolean, System.Char, System.Double, System.Float, System.Int32, System.Int64 und Strukturen (structs).
Der restliche Speicher wird für den sogenannten **Managed Heap (verwaltete halde)**

[12] .NET Anwendungen verwenden wie Win32 Anwendungen das Win32 Portable Executable Dateiformat. Bei diesem Format wird im IMAGE_OPTIONAL_HEADER die Stackgröße codiert. Weitere Informationen finden Sie unter `http://msdn.microsoft.com/en-us/library/ms809762.aspx`

[13] Die .NET Framework 4.0 Funktionalität ist beschrieben auf `http://msdn.microsoft.com/en-us/library/5cykbwz4.aspx`

benutzt. Dort werden alle Referenztypen abgelegt. Die CLR unterteilt den Heap in Bereiche mit unterschiedlicher Größe (Paging). Sollte ein neuanzulegendes Objekt nicht in den aktuellen Heap hineinpassen, wird vom Betriebssystem ein neuer Speicherbereich (page) angefordert. Somit wächst der Heap bis zu seiner maximalen Größe. (Adressraum - Codebereich - Konstantenbereich - Stack = max. Heapgröße).
Der **Garbage Collector (Müllsammler)** ist die Komponente der Speicherverwaltung, die für das Aufräumen des Speichers zuständig ist. Der GC arbeitet mit Generationen:

- Gen0 "neu oder kurzlebig" - hier liegen fast alle neu erstellten Objekte - die Generation ist ungefähr 256 KB groß.
- Gen1 "mittelfristig" - hier befinden sich Objekte die *eine* Garbage Collection überlebt haben - die Größe beträgt ca. 2MB.
- Gen2 "langlebig"- Objekte die mehr als eine Garbage Collection überstanden haben befinden sich in dieser Generation - sie hat eine Größe von ca. 10MB.

Die Größen von Generationen können nur ungefähr angegeben werden, da das .NET zur Laufzeit diese Größen optimiert. Die Größe der Gen2 wächst mit der Laufzeit des Programms bis zur maximalen Größe die annähernd der gesamten Größe des Heaps entspricht (abzüglich Gen0 und Gen1). Im Programmcode kann die Generation, in welcher sich ein Objekt gerade befindet, über eine static Methode der Klasse GC ausgelesen werden.

```
1 int GetGeneration(object obj)
```

Listing 2.14: Methode zum Auslesen der aktuellen Heap-Generation eines Objektes.

Sollte der GC ermitteln, dass viele kurzlebige Objekte angelegt werden, wird der GC wahrscheinlich die Gen0 vergrößern. Dadurch kann mit weniger Garbage Collections wieder viel Speicher freigegeben werden. Es gibt verschieden Gründe, welche eine Garbage Collection veranlassen können:

1. Gen0, Gen1 oder Gen2 sind voll
2. es wird im Programmcode explizit aufgerufen

```
1     GC.Collect()
```

Listing 2.15: Erzwingen einer Garbage Collection.

3. auf dem Heap ist kein Platz mehr um ein neues Objekt anzulegen

Eine Garbage Collection läuft in zwei Schritten ab. Die erste Phase (Mark) markiert alle Objekte die nicht mehr benötigt werden. Im zweiten Schritt (Sweep) wird der Speicher der markierten Objekte freigegeben. Zu einer Fragmentierung der 0. und 1. Generation kann es bei diesem Entwurf nicht kommen. Ein Objekt wird entweder in eine höhere Generation verschoben (von 0 nach 1 bzw. von 1 nach 2) oder gelöscht. Die 2.Generation wird defragmentiert, sobald die CLR Speicherplatz benötigt. Dafür muss der laufende

Prozess allerdings gestoppt werden, was das Programm langsamer erscheinen lässt. Um zu entscheiden, ob ein Objekt noch benötigt wird, sucht der GC Wurzeln (Roots). Wurzeln können statische Felder, Methodenparameter, lokale Variablen oder CPU Register sein. Hat ein Objekt (oder eine Gruppe von Objekten) keinen Verweis auf eine Wurzel ist es aus dem Programmcode nicht mehr zugreifbar und kann aus dem Speicher entfernt werden. Eine Besonderheit ist der **Large Object Heap (LOH)**. Auf diesen werden alle Objekte, die größer als 85KB sind, gespeichert.

```
1  byte[] bigData = new byte[85 * 1024];
```

Listing 2.16: Beispiel eines Objektes, das auf dem LOH gespeichert wird.

LOH Objekte werden schon zum Zeitpunkt der Erstellung in der 2. Generation gelistet, da davon ausgegangen werden kann, dass so große Objekte länger bearbeitet werden. Der LOH sorgt in einem in komplexen Softwaresystemen für Probleme, da keine Defragmentierung stattfindet! Durch ungünstige Fragmentierung kann es zu einer Ausnahme "Speicher nicht ausreichend" (out of memory exception) kommen, obwohl weniger als 10 % des zur Verfügung stehenden Speichers ausgenutzt sind. Im Anhang (155MB für Out Of Memory) ist ein Beispielcode für eine falsche Nutzung des LOH aufgeführt.

2.4.2 Wachstumsverhalten von Listen

In komplexen Softwaresystemen ist die Verwaltung von vielen Objekten unerlässlich. Wer dazu nur Standardklassen des .NET verwendet, kann Probleme bekommen. Die Klasse List<T> arbeitet intern auf einem Array vom generischen Type T. In diesem Array werden entweder direkt die Daten gespeichert (bei Value-Types) oder es werden Referenzen auf entsprechende Objekte (4 Byte groß bei 32-Bit-Betriebssystem) gespeichert (bei Reference-Types). Das Array, welches die Klasse List<T> verwendet, wird ab einer bestimmten Anzahl an Einträgen auf dem LOH abgelegt und zu dessen Fragmentierung beitragen.

$$\frac{85 \cdot 1024}{4} = 21760$$

21760 Einträge klingt erstmal viel, doch hier kommt das Wachstumsverhalten der Listen ins Spiel. Die Größe des Arrays wird nicht mit jedem Element vergrößert, das in die Liste hinzugefügt wird (das würde zu viel Laufzeit benötigen). Sobald ein Element hinzugefügt werden soll, welches nicht mehr in das Array passt, wird die Größe des Arrays verdoppelt. Es sind also nur ca. **10880 Einträge** verfügbar, bevor die Daten auf dem LOH abgelegt werden. Wenn man ungefähr abschätzen kann, wie viele Elemente eine Liste beinhalten wird, empfiehlt es sich bereits beim Erstellen der Liste diese Größe anzugeben. Dadurch werden viele unnötige Größenänderungen des Arrays vermieden.

```
1    //we know that more than 7000 elements will be stored in this list
2    IList<Foobar> myList = new List<Foobar>(8000);
```

Listing 2.17: Richtiges Anlegen einer Liste, bei abschätzbarer Anzahl der Elemente.

Um dieses Problem zu reduzieren, sollte schon beim Entwurf darauf geachtet werden, dass derjenige, welcher verschiedenen Berechnung auslöst, schon die benötigten Objekte in die Berechnungen übergibt (anstatt für jede Berechnung die Daten neu zusammenzusammeln). Eine mögliche Lösung wäre es auch, selbst implementierte Datenstrukturen wie eine doppelt verkette Liste zu implementieren. Da diese nur aus vielen Knoten von rund 16/20 Byte (4 Byte Next-Referenz, 4 Byte Previous-Referenz, 4 Byte Referenz auf Wert bzw. 1 bis n Byte bei Werttypen) besteht, wird sie nicht auf dem LOH gespeichert. Das Kapitel Einfache LOH freie Collection zeigt beispielhaft die Implementierung einer solchen Liste.

2.4.3 Eventsystem

In .NET gibt es zur Benachrichtung anderer Objekte die Events. Ein Event basiert auf einem MultiCastDelegate. Dieser arbeitet intern mit einer **stark referenzierten** (strong referenced) Liste von Zielen. Das heißt Objekte, die sich auf dieses Event anmelden bleiben so lange am Leben bis:

- sie sich wieder abmelden
- das Objekt, an dem Sie sich angemeldet haben, freigegeben wird

Als Beweis soll eine kleine Anwendung dienen:

```
1  class Program {
2    static void Main(string[] args) {
3      Manager m = new Manager();
4      long counter = 1;
5      while (Console.ReadKey().Key != ConsoleKey.Escape) {
6        //clean up memory
7        GC.Collect();
8        GC.WaitForPendingFinalizers();
9        //create local variable
10       NotOkay no = new NotOkay(m);
11       no.DoSomething();
12       if (counter % 10 == 0) {
13         m.RiseEvent();
14         Console.WriteLine("GC run {0} times",
15           GC.CollectionCount(0));
16       }
17       ++counter;
```

```
18      } //end of block, all local variables can be free
19    }
20  }
21  internal class Manager {
22    public delegate void Foobar();
23    public event Foobar OnFoobar;
24
25    public void RiseEvent() {
26      if (OnFoobar != null) OnFoobar.Invoke();
27    }
28  }
29  internal sealed class NotOkay {
30    private Manager m_Manager;
31    public NotOkay(Manager manager) {
32      m_Manager = manager;
33      m_Manager.OnFoobar += () =>
34      {
35        Console.WriteLine("{0} - do much more things",
36          GetHashCode());
37      };
38    }
39    public void DoSomething() {
40      Console.WriteLine("{0} - do some special things",
41        GetHashCode());
42    }
43  }
```

Listing 2.18: Anwendung für die Erhaltung alle Objekte durch Eventanmeldungen.

Dieses kleine Programm zeigt folgende Ausgabe:

```
62476613 - do some special things
11404313 - do some special things
64923656 - do some special things
44624228 - do some special things
17654054 - do some special things
52727599 - do some special things
14347911 - do some special things
51393439 - do some special things
26756241 - do some special things
23264094 - do some special things
62476613 - do much more things
11404313 - do much more things
64923656 - do much more things
44624228 - do much more things
```

```
17654054 - do much more things
52727599 - do much more things
14347911 - do much more things
51393439 - do much more things
26756241 - do much more things
23264094 - do much more things
GC run 10 times
```

Am Ende der While-Schleife wird die Instanz von der Klasse NotOk nicht mehr benötigt. Der GC hat aber keine Möglichkeit, diese Instanz wieder freizugeben, da sie durch die Eventanmeldung stark referenziert wird. Je nach Verwendung ist dies eine Möglichkeit um den Lebenszyklus von Objekten zu beeinflussen oder ein potenzielles Speicherproblem.
Ein Vorteil ist, dass ein Controller sich nur auf die Events eines UI-Controls anmeldet für die er zuständig ist. Er bleibt damit am Leben, solange die UI-Controls nicht vom GC entsorgt werden und kann seine Aufgaben erledigen. Dabei hat das UI-Control keine Kenntnis von dem Controller. Das UI-Control kann in anderen Oberflächen auch von anderen Controllern verwendet werden ohne dass der Quelltext des UI-Controls angepasst werden muss.
Das ist eine **lose Kopplung (loosely coupled)**. Diese hat in der Software-Architektur den Vorteil, dass z.B. die Änderung im UI-Control keine Änderung im Controller nach sich zieht, es sei denn die Signatur des Events ist betroffen. Der Architekturstil *Implicit Invocation* beschreibt die Entkopplung von Objekten mittels Events. Durch Entkoppelung werden Abhängigkeiten minimiert. Dadurch ist es einfacher die Software zu pflegen. Diese Eigenschaft machen sich Event-Based-Components[14] zunutze.
Eine negative Variante wurde im Beispiel am Anfang des Kapitels gezeigt. Wenn ein Objekt sich nicht von einem Event abmeldet kommt es dazu, dass viel mehr Objekte als benötigt im Speicher gehalten werden.

2.4.4 Lebenszyklus von Objekten

Der normale Lebenszyklus ist in .NET sehr einfach: Ein Benutzer löst eine Aktion aus (Öffnen eines grafischen Editors).

⇓

Die Oberfläche lädt das entsprechende DomainObject.

⇓

Das DomainObject lädt weitere Objekte. DomainObjekte sind oft Container für weitere Objekte. Ein Computer kann auch als Container bezeichnet werden. Er enhält Festplatten, Mainboard, Speicher, Grafikkarte.

⇓

[14] Genauere Informationen über EBC finden Sie unter `http://www.heise.de/developer/artikel/Event-Based-Components-Architektur-von-Software-in-neuem-Licht-1240993.html`

Ein DomainObject lädt einen Service oder registriert sich an einer anderen Komponente für Änderung.

⇓

Der Benutzer arbeitet mit dem DomainObject und die geladenen Objekte verrichten ihre Aufgaben.

⇓

Ein Benutzer löst eine neue Aktion aus (Schließen eines grafischen Editors).

⇓

Die Oberfläche setzt seine Referenz des DomainObjects auf null.

⇓

Der Garbage Collector gibt den gesamten Speicher der Objekt-Wolke frei, sobald eine Garbage Collection erfolgt.

Eine Programmierrichtlinie bei C# ist "Für jede **Registrierung (+=)** muss es auch eine **Deregistrierung (-=)** geben". Im oberen Lebenszyklus fehlt jedoch ein Schritt, in dem alle DomainObjects sich deregistrieren können, deshalb würde das Objekt (A), welches sich *außerhalb* der Objekt-Wolke an einem statischen Dienst registriert hat, von der Garbage Collection nicht abgeräumt werden. Dies bezieht sich auch auf alle Objekte (B, C, D), die sich bei A registriert haben. Anstatt die Referenz des DomainObjects einfach nur auf null zu setzen, müsste es vorher eine Art "saubermachen" (Clean Up) geben. Dies ist unabhängig davon, ob für das Aufräumen ein spezielles Interface vorgesehen ist, in der Basisklasse aller Objekte eine abstrakte Methode "Unload" einführt oder die Schnittstelle **IDisposable**, welche im Kapitel 5.1.1 behandelt wird, dafür verwendet. Jedes Objekt muss eine Methode zum Aufräumen haben und diese muss auch aufgerufen werden. Wichtig ist, dass ein Objekt den Aufräumaufruf an die von ihm erstellten Objekte weiterdelegiert.

2.4.5 Zeichenketten

Zeichenketten (strings) werden in jedem Programm exzessiv verwendet. Ein paar Verwendungsmöglichkeiten sind:

1. Anzeige von Texten an der Oberfläche
2. Hantierung von Pfadangaben
3. Nutzung als Konstanten für Reflektion
4. Erzeugung und Bearbeitung von für Menschen lesbare Dateien
5. Tracing und Logging

Zeichenketten sind unveränderlich. Daraus folgt, dass jede Veränderung an einem String zur Folge hat, dass ein weiterer (modifizierter) String im Speicher kreiert wird. Strings werden von der CLR auf dem Heap gespeichert, da sie von System.Object erben und somit Referenztypen sind. Zeichen innerhalb eines Strings werden als System.Char (16

Bit Unicodezeichen) gespeichert. Strings einer bestimmten Größe werden von der CLR auf den LOH abgelegt. Sie können damit auch zur Fragmentierung des LOH beitragen. Das folgende Listing beinhaltet offensichtliche und nicht offensichtliche "Speichersünden":

```
1  public class Bad
2  {
3    static void Main(string[] args)
4    {
5      string hallo = ". Hallo";
6      string welt = " Welt";
7      string text = hallo + welt;
8
9      for (int i = 1; i < 6; i++)
10     {
11       Console.WriteLine(i + text);
12     }
13   }
14 }
```

Listing 2.19: Schlechtes Beispiel für verarbeitung von Zeichenketten.

Tabelle 2.2[15] zeigt, welche Strings sich nach Beendigung der Schleife und vor einer Garbage Collection im Speicher befinden:

[15] Auszug vom Befehl "!dumpheap -strings" der sos.dll. Weitere Informationen finden Sie unter `http://msdn.microsoft.com/en-us/library/bb190764.aspx`

Anzahl	Totale Größe	Wert
1	28	". Hallo"
1	24	" Welt"
1	40	". Hallo Welt"
2 [A]	32	"1"
1	40	"1. Hallo Welt"
2 [A]	32	"2"
1	40	"2. Hallo Welt"
2 [A]	32	"3"
1	40	"3. Hallo Welt"
2 [A]	32	"4"
1	40	"4. Hallo Welt"
2 [A]	32	"5"
1	40	"5. Hallo Welt"
	452	

Tabelle 2.2: Auflistung aller Strings im Speicher des "Hallo Welt"-Programms

[A] Die Zeichenketten "1", "2", "3" befinden sich bereits zweimal und "4", "5" einmal im Speicher eines .NET Programmes, ohne dass das die String geladen hat. Diese Standard-Strings wurden herausgerechnet.

Um speichereffizient zu programmieren sollten Modifikationen an Strings vermieden werden:

```
1  public class LessMemory
2  {
3    static void Main(string[] args)
4    {
5      string text = ". Hallo Welt";
6      for (int i = 1; i < 6; i++)
7      {
8        Console.Write(i);
9        Console.WriteLine(text);
10     }
11   }
12 }
```

Listing 2.20: Zeichenkettenverarbeitung mit weniger Speicherverbrauch.

Tabelle 2.3 zeigt den benötigten Speicher nach Speicherverbrauchoptimierungen.

Anzahl	Totale Größe	Wert
1	40	". Hallo Welt"
2 [A]	32	"1"
2 [A]	32	"2"
2 [A]	32	"3"
2 [A]	32	"4"
2 [A]	32	"5"
	200	

Tabelle 2.3: Auflistung aller Strings im Speicher nach Speicher-Optimierung des "Hallo Welt"-Programms

[A] Die Zeichenketten "1", "2", "3" befinden sich bereits zweimal und "4", "5" einmal im Speicher eines .NET Programmes, ohne dass das die String geladen hat. Diese Standard-Strings wurden herausgerechnet.

Durch die Optimierung konnten 252 Byte Speicher zur Laufzeit gespart werden, allerdings hat sich die Laufzeit des Programmes verschlechtert. Die Laufzeit des ersten Programms beträgt im Durchschnitt 2580 Ticks für die Ausführung der Schleife. Beim zweiten Programm werden durchschnittlich 2920 Ticks verbraucht. Eine dritte Variante des Programmes (welche auf Laufzeit optimiert ist) benötigt durchschnittlich nur 2350 Ticks, verbraucht dabei aber signifikant mehr Speicher.

```
1  public class MoreSpeed
2  {
3    static void Main(string[] args)
4    {
5      string text = ". Hallo Welt";
6      var sb = new StringBuilder();
7      for (int i = 1; i < 6; i++)
8      {
9        sb.AppendFormat("{0}{1}\n", i, text);
10     }
11     Console.WriteLine(sb.ToString());
12   }
13 }
```

Listing 2.21: Schnellste Variante zur Verarbeitung von Zeichenketten.

Tabelle 2.4 zeigt den benötigten Speicherverbrauch nach Laufzeitoptimierung.

Anzahl	Totale Größe	Wert
1	40	". Hallo Welt"
2 [A]	32	"1"
2 [A]	32	"2"
2 [A]	32	"3"
2 [A]	32	"4"
2 [A]	32	"5"
5	780	" 1.Hallo Welt\n2.Hallo Welt\n 3.Hallo Welt\n4.Hallo Welt\n5.Hallo Welt\n"
	940	

Tabelle 2.4: Auflistung aller Strings im Speicher nach Laufzeit-Optimierung des "Hallo Welt"-Programms

[A] Die Zeichenketten "1", "2", "3" befinden sich bereits zweimal und "4", "5" einmal im Speicher eines .NET Programmes, ohne dass das die String geladen hat. Diese Standard-Strings wurden herausgerechnet.

Wie die drei Programme zeigen, ist es wichtig zu wissen, was der Use Case bei der Verwendung von Strings ist.
In der CLR existiert ein Mechanismus um Speicher zu sparen und Vergleiche von Strings effektiv durchzuführen - das String-Interning. Jeffrey Richter beschreibt es wie folgt (aus [Richter, Seite 274ff]):

> "Falls sie mehrere Instanzen des selben Strings mehrfach im Speicher haben, verschwenden Sie Speicherplatz, weil Strings unveränderlich sind. Die Speichernutzung ist viel effizienter, wenn es nur eine Instanz des Strings im Speicher gibt und alle Variablen, die auf diesen String verweisen, einfach auf dasselbe String-Objekt zeigen.
> Falls Ihre Anwendung öfter mit einem numerischen Vergleich unter Beachtung der Groß- und Kleinschreibung überprüfen muss, ob Strings gleich sind, oder falls Sie erwarten, dass viele string-Objekte denselben Inhalt haben, können Sie die Leistung deutlich steigern, indem sie vom Interning-Mechanismus der CLR Gebrauch machen. Wenn die CLR initialisiert wird, legt sie eine interne Hashtabelle an. Die Schlüssel dieser Hashtabelle sind Strings, die Werte sind Verweise auf String-Objekte im verwalteten Heap. [...]
> Beachten Sie, dass der Garbage Collector keine Strings beseitigen kann, auf die in der internen Hashtabelle verwiesen wird. Die Hashtabelle speichert nämlich noch gültige Verweise auf diese String-Objekte. String-Objekte, auf die in der Hashtabelle verwiesen wird, können erst dann freigegeben werden, wenn die AppDomain entladen oder der Prozess beendet wird."

Für eine Software, welche mit häufigen String-Vergleichen arbeitet, ist Interning eine Möglichkeit, die Laufzeit zuverbessern. Da die zeitaufwendigen Stringvergleiche sehr

schnell werden. Dafür wird der Heap dauerhaft mit Strings belegt.

```
1  public class StringComparer : IEquatable<StringComparer>
2  {
3    private string Text {get; private set; }
4    public StringComparer (string text)
5    {
6      Text = string.IsInterned(text);
7      if (Text == null)
8      {
9        Text = string.Intern(text);
10       text = null;
11     }
12   }
13   public bool Equals(StringComparer other)
14   {
15     return object.ReferenceEquals(Text, other.Text);
16   }
17 }
```

Listing 2.22: Verwendung von String.Interning um Vergleiche von Zeichenketten zu beschleunigen.

Das Kapitel String interning ohne dauerhaften Speicherverbrauch zeigt eine einfache Implementierung, die den Interning Mechanismus nachbildet, jedoch ohne den Speicher dauerhaft zu belegen. Durch die Verwendung von WeakReferences brauchen die Strings nicht über den Lebenszyklus der AppDomain bzw. des Programmes im Speicher gehalten werden.

2.4.6 Nicht verwaltete Ressourcen

Nicht verwaltete Ressourcen (unmanged ressources) können aufgrund von Implementierungsdetails des .NET Frameworks Probleme verursachen. Diese Probleme sollen anhand der Klasse Bitmap besprochen werden. Als Illustration soll ein Programm dienen. In dieser Anwendung werden in einer ListView Bilder angezeigt. Da die anzuzeigenden Bilder alle eine Größe über 85KB besitzen, sind die Bitmap-Objekte auf dem LOH gespeichert. Damit es zu keiner Fragmentierung des LOH oder zu Speichermangel kommt, besitzt die ListView einen virtuellen Modus und gibt die Bilder, sobald sie außerhalb des sichbaren Bereiches sind, frei. Um Laufzeit und Speicher einzusparen, werden die Bilder in einem Bilderpool verwaltet. Dieser besitzt WeakReferences auf die Bitmap-Objekte. Sollte ein Bild angefordert werden, welches sich im Speicher befindet, wird die Referenz zurückgegeben, andernfalls wird das Bitmap geladen.

```
public class PicturePool
{
  private Dictionary<string, WeakReference> m_Pool;
  public PicturePool()
  {
    m_Pool = new Dictionary<string, WeakReference>();
  }
  public Bitmap this[string path]
  {
    get
    {
      if(String.IsNullOrEmpty(path))
        throw new ArgumentNullException("path");
      return GetPicture(path);
    }
  }
  private Bitmap GetPicture(string path)
  {
    if(!m_Pool.ContainsKey(path)
      || !m_Pool[path].IsAlive
      || m_Pool[path].Target == null)
    {
      m_Pool[path] = new WeakReference(new Bitmap(path));
    }
    return m_Pool[path].Target as Bitmap;
  }
}
```

Listing 2.23: Cache zum Verwalten von Bitmap-Objekten.

Beim Scrollen in der ListView kommt es zu einer ArgumentException "Ungültiger Parameter". Der PicturePool hat die Referenz auf ein Bitmap zurückgeliefert, welches gerade abgeräumt wird. Diesen Status nennt man "Disposing". Nachdem das Objekt abgeräumt wurde, heißt der Status "Disposed". In beiden Fällen sollte mit dem Objekt nicht mehr gearbeitet werden. Eine Möglichkeit herrauszufinden, ob ein Objekt gerade abgeräumt wird, gibt es nicht.
Bei eigenen Klassen bietet es sich an, eine *public Property bool Disposed{ get; private set;}* zu implementieren, welche beim Betreten der Dispose Methode auf true gesetzt wird. In jeder anderen Methode sollte die Disposed Property abgefragt werden. Ist sie gesetzt muss eine ObjectDisposedException geworfen werden.
Bei Klassen einer vordefinierten Bibliothek funktioniert dies nicht. Eine einfache Lösung ist ein Wrapper um das Bitmap Objekt (eine Ableitung kann nicht in Betracht gezogen werden, da die Klasse Bitmap *Sealed* ist). Der PicturePool gibt nur Referenzen auf einen BitmapWrapper heraus und fragt am BitmapWrapper an, ob er Disposed ist. Wenn ja,

muss der Wrapper für dieses Bitmap neu geladen werden.

```
public class PicturePool2
{
  private Dictionary<string, WeakReference> m_Pool;
  public PicturePool2()
  {
    m_Pool = new Dictionary<string, WeakReference>();
  }
  public BitmapWrapper this[string path]
  {
    get
    {
      if(String.IsNullOrEmpty(path))
        throw new ArgumentNullException("path");
      return GetPicture(path);
    }
  }
  private BitmapWrapper GetPicture(string path)
  {
    if(!m_Pool.ContainsKey(path)
      || !m_Pool[path].IsAlive
      || m_Pool[path].Target == null)
    {
      m_Pool[path] = new WeakReference(new BitmapWrapper(path));
    }
    else
    {
      BitmapWrapper pic = m_Pool[path].Target as BitmapWrapper;
      if(pic.Disposed)
      {
        m_Pool[path] = new WeakReference(new BitmapWrapper(path));
      }
    }
    return m_Pool[path].Target as BitmapWrapper;
  }
}
public sealed class BitmapWrapper : IDisposable
{
  public BitmapWrapper(string path)
  {
    if(String.IsNullOrEmpty("path"))
      throw new ArgumentNullException("path");
    Bitmap = new Bitmap(path);
  }
  public bool Disposed { get; private set; }
```

```
45    public Bitmap Bitmap { get; private set; }
46    public void Dispose()
47    {
48      Disposed = true;
49      Bitmap.Dispose();
50    }
51  }
```

Listing 2.24: Cache zum Verwalten von Bitmap-Wrapper-Objekten.

Dies ist jedoch keine allgemeingültige Lösung. Sollte jemand BitmapWrapper.Bitmap.Dispose() aufrufen, kommt es wieder zu dem selben Problem. Für viele Szenarien ist diese Variante dennoch absolut ausreichend. Die beste Lösung ist es einen kompletten Wrapper für das Bitmap zu entwerfen, dieser:

1. leitet sich von System.Drawing.Image ab und ist sealed.
2. beinhaltet einen privaten Member vom Typ System.Drawing.Bitmap.
3. implementiert alle public Methoden von Bitmap und Image und leitet diese an den privaten Member weiter (notfalls mit new Überschreiben).
4. implementiert IDisposable und die public Property Disposed.

Einen kompletten Wrapper zu erstellen ist aufwändig, jedoch lohnt es sich gerade bei größeren Projekten. Im Umfeld von .NET besteht o.g. Problem zusätzlich zu den anderen (z.B. Memory Leaks und Dead Locks). Daher ist es sinnvoll in Verbindung mit unmanged Ressourcen immer Wrapper vorzusehen.

2.5 Probleme komplexer Softwaresysteme

Was bei Softwaresystemen normaler Größe eine optimale Lösung darstellt, kann bei großen Softwaresystemen zu Problemen führen. Dieses Kapitel dient als Sensibilisierung für dieses Thema.

2.5.1 Massenoperationen

Hinter dem Begriff "Massenoperationen" (Bulk Operations) verbirgt sich eine Metapher. Sie wird beschrieben durch die Frage: *"Kommt meine Architektur/Implementierung mit 500000 Objekten zurecht?"* Diese Frage soll anhand eines Beispieles beleuchtet werden:

> "Es existiert ein Dialog, welcher in einem Baum (sortiert nach Namen) Computer anzeigt. Wenn im Baum ein Computer ausgewählt wird, zeigt der Dialog zusätzliche Informationen zu diesem Computer an (DNS-Namen, IP-

> Adresse, MAC-Adresse u.ä.). Das Baum-Oberflächenelement arbeitet intern mit einer sortierten Liste. Da es keine Hinzufügen-Methode für Massenoperationen gibt, wird mehrfach ein einzelner Computer hinzugefügt. Dies funktioniert für Klasse C Subnetze (z.B. 192.168.0.0) wunderbar. Nun sollen aber alle Elemente eines Klasse A Subnetzes (z.B. 10.0.0.0) angezeigt werden. Es dauert fast eine Stunde, bis sich der Dialog öffnet. Das Problem besteht darin, das mit jedem hinzufügen eines Elementes die interne Liste sortiert wird. Würden erst alle Elemente hinzugefügt und dann die Liste einmal am Ende sortiert, ginge das Öffnen in ein paar Sekunden."

Dies ist kein einfacher Programmierfehler, sondern eine Schwäche der Architektur.

- Warum gibt es keine Schnittstelle, die eine Liste oder ein Array von Computern als Parameter akzeptiert?
- Warum gibt es keinen virtuellen Modus, der erstmal nur die Elemente lädt, die auf einmal angezeigt werden können?

Beim Entwurf der Schnittstellen und Methodensignaturen sollte man darauf achten, dass Methoden, soweit sinnvoll, immer eine Menge von Elementen übergeben bekommen. Natürlich muss die Möglichkeit bestehen, dass diese Menge nur ein Element enthält. Das trifft besonders auf Schnittstellen zwischen Schichten oder Komponenten zu. Beispielsweise ist eine Persistenzschicht, die nur das Lesen/Schreiben einer Eigenschaft eines Objektes zulässt, für sehr viele Use-Cases ungeeignet, da oft mehrere Eigenschaften benötigt werden.
Sehr häufig kommt es bei Massenoperation zu Speicherengpässen. Solche Probleme zeigen Fehler in der Architektur auf. Meistens werden alle Objekte im Speicher gehalten, die dem Benutzer angezeigt werden oder die er selektiert hat. Dieser Tatsache wird bei vielen Architekturentscheidungen nicht widersprochen. Sobald es jedoch die Funktionalität "Alles markieren" gibt kann es sehr schnell zu Laufzeit- und Speicherproblematiken kommen. Sinnvoller ist es, nur anzuzeigen dass alles markiert ist und die Domänen-Objekte erst zu laden, wenn der Benutzer eine Aktion (wie Kopieren) mit ihnen ausführt. Beim Abarbeiten der Benutzeraktion sollte man die Selektion in Cluster einteilen, so dass z.B. immer nur 50 Objekte geladen und bearbeitet werden. In komplexen Softwaresystemen kommt es früher oder später immer zu Problemen mit Massenoperationen. Dann ist es jedoch meistens zu spät um dan ganzen Entwurf nochmal anzupassen. Daher sollte diese Problematik gleich von Anfang an (z.B. in Form einer Cluster-Komponente) betrachtet werden.
In komplexen Softwaresystemen kommt man meistens zwangsläufig in die Situation, dass alter Quellcode und Komponenten (Legacy Code) aus früheren Projekten wiederverwendet werden sollen. Ein typisches Beispiel dafür sind ActiveX-Controls. Jede Instanz eines ActiveX-Controls benötigt ein eigenen GDI-Handle. Man stelle sich nun vor, dass in einer Tabelle 500.000 Zeilen enthalten sind und jeder dieser Zeilen eine Zelle enthält die eine Instanz dieses ActiveX-Controls hostet. Wenn nun alle Zeilen auf einmal

geladen werden kommt man sehr schnell an die Grenzen des Betriebssystems. In dem gewählten Beispiel bräuchten wir somit 500.000 GDI-Handles. Da bei Windows aktuell die maximale Anzahl an GDI-Handles bei 65.536[16] liegt hätten wir in diesem Beispiel die Grenze des Betriebssystems erreicht und unsere Applikation könnte die Tabelle nicht mehr anzeigen. Die praktische Grenze liegt sogar noch weit darunter. Windows XP, Vista und Windows 7 steuern über die Registry, wie viele GDI-Handles ein Prozess bekommen kann. Die Standardeinstellung liegt bei gerade mal 10.000 GDI-Handles. Da nicht nur die Tabelle sondern auch Menüs, Buttons ein GDI-Handle benötigen, erreicht man diese Grenze sehr schnell. Daher sollten Komponenten so designed sein, dass sie mit möglichst wenig GDI-Handles auskommen. Dieses Verfahren nennt sich Windowless Architektur. In unserem Beispiel bedeutet dies, dass für die gesamte Tabelle ein einziges GDI-Handle benötigt wird und nur für die jeweils aktuell sichtbaren Zellen die ein ActiveX-Control hosten ein eigenes GDI-Handles erzeugt.
Bei Datenstrukturen und Dateiformaten ist die Sichtweise der Massenoperationen besonders interessant. Ein UInt16-Wert zur Adressierung von Knoten in einem Datengraphen kommt maximal mit 65536 Knoten zurecht. Eine als einfach verkettete Liste realisierte Datenstruktur ist akzeptabel, wenn immer alle Objekte der Datenstruktur verwendet werden müssen. Wenn wahlfreier Zugriff benötigt wird, eignet sich ein Array oder eine Hash-Tabelle besser. Auch wenn ein UInt64-Wert zur Adressierung verwendet wird, sollte man immer mindestens das MostSignificanBit (MSB) reservieren. Falls man doch mit mehr Elementen arbeiten muss, kann zum Beispiel definiert werden, dass bei gesetztem MSB zusätzlich ein weiteres UInt64 zur Adressierung hinzugezogen wird (dessen MSB wird natürlich auch wieder reserviert).

2.5.2 Metadateien

Informationen zu Domänen-Objekten werden oft in Beschreibungsdateien (auch Metadateien genannt) abgelegt. Das zurzeit gebräuchlichste Format ist XML. Damit Eigenschaften, die bei mehrere DomainObjekte identisch sind, nicht in jeder XML-Datei stehen, gibt es eine weitere XML-Datei, die diese enthält. Somit ergibt sich ein Baum von Dateien, die benötigt werden, um aller Informationen zu einem DomainObjekt auszulesen. Je komplizierter (ausmodellierter) das DomainModell ausfällt, desto mehr Dateien sind es. Die Zeitdauer zum Laden alle Informationen liegt nicht unbedingt daran, das XML im Vergleich zu proprietären Dateiformaten sehr aufgebläht ist, sondern das Laden einzelner Dateien von der Festplatte ist zeitintensiv. Eine große Datei (wie ein Archiv) zu laden, geht sehr viel schneller, auch wenn die Daten dearchiviert werden müssen. Es dient auch dem Know-how-Schutz, wichtige Informationen nicht für Menschen lesbar auf der Festplatte abzulegen.

[16] siehe `http://msdn.microsoft.com/en-us/library/ms724291%28VS.85%29.aspx`

3 Unterscheidung von komplexen Softwarearchitekturen

Die Unterscheidung von Softwarearchitekturen unterstützt den Softwarearchitekten, indem "Kernentscheidungen" thematisiert werden. In diesem Kapitel werden einige Unterscheidungskriterien vorgestellt und über Ihren strategischen Nutzen für die Softwarearchitektur diskutiert.

3.1 Offene und geschlossene Architekturen

Die Unterschiede zwischen offener und geschlossener Architektur lassen sich am einfachsten anhand von Hardware erklären. Details einer offenen Hardware-Architektur sind beschrieben und frei verfügbar. Die vorhandene Architektur kann um eigene Bedürfnisse erweitert werden. Im Gegensatz dazu steht die geschlossen Architektur (proprietäre Architektur). Diese legt die Details nicht offen und kann damit nur von "Insidern" erweitert werden.
In der Softwareentwicklung liegt die Annahme nahe, dass offene Architekturen die Integration von Drittanbieter Software (thrid party tools) unterstützen und geschlossene Architekturen nicht. In Wirklichkeit sind die Grenzen aber fließend: das IPhone OS ist via Apps erweiterbar, doch da nur solche Apps im Appstore erhältlich sind, die von Apple zertifiziert wurden, ist sie wesentlich geschlossener als das Android OS. Bei Android kann jeder Entwickler seine Apps im Appstore anbieten.
Die Abbildung für offene und geschlossene Architekturen in der Softwareentwicklung, welche wirklich relevant ist, definiert die **interne Sicht**. Abbildung 3.1 zeigt ein einfaches Szenario, indem eine Wertänderung (vom Benutzer) persistiert werden soll.
Das Sequenzdiagramm 1, in Abbildung 3.2, soll die geschlossene Architektur darstellen. Es ist zu sehen, dass die Oberfläche nur direkt in ihrer und der ihr unterlagerten Schicht (Geschäftslogik) arbeitet. Die Geschäftslogik arbeitet ebenfalls nur in ihrer Schicht und der ihr unterlagerten Schicht.
Das Sequenzdiagramm 2, in Abbildung 3.3, soll die offene Architektur darstellen. Die Oberfläche greift direkt auf die Persistenzschicht zu und umgeht die Geschäftslogik-Schicht. Bei der offenen Architektur hätte der Ablauf identisch zu dem bei der geschlossenen Architektur sein können, aber es muss eben nicht.
Natürlich ist es im Beispiel der offenen Architektur möglich, das Szenario korrekt abzubilden, z.B. hätte das Geschäftslogikobjekt sich auf Wertänderung bei der Persistenz anmelden können und würde die Änderung dadurch mitbekommen. In der Regel muss eine offene Architektur mit mehreren Varianten des Workflows umgehen können als eine geschlossene Architektur.
Tabelle 3.1 zeigt kurz und prägnant die Vor- und Nachteile der zwei Varianten.

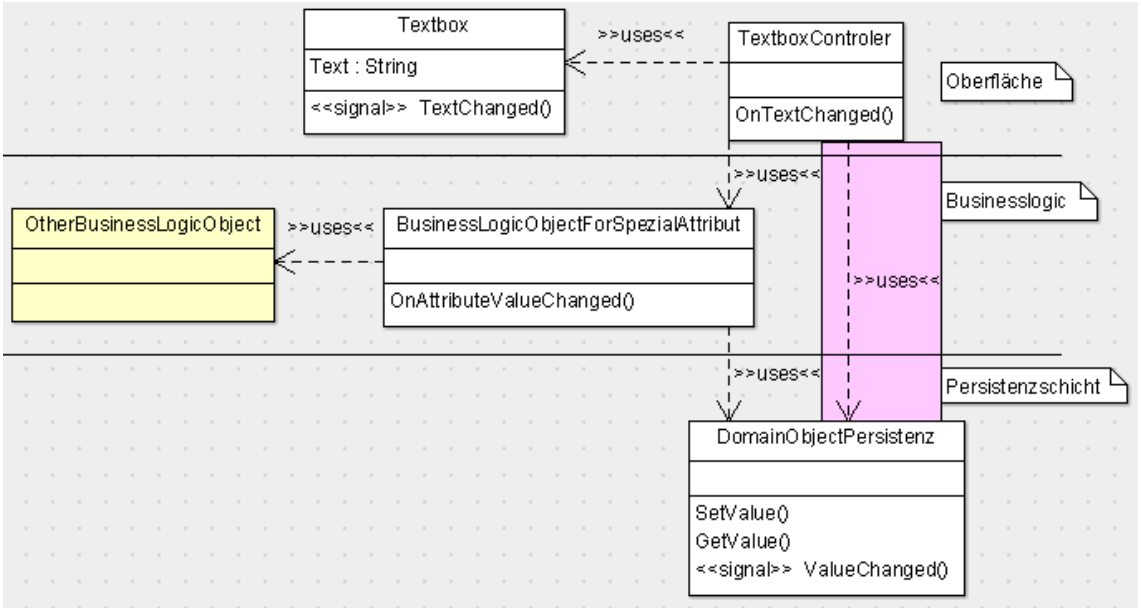

Abbildung 3.1: Klassendiagramm - Oberflächenänderung persistieren

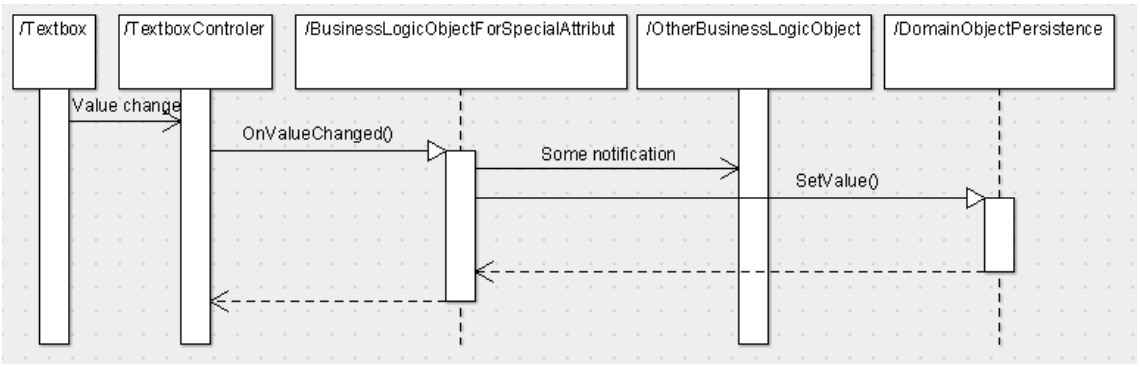

Abbildung 3.2: Sequenzdiagramm - Oberflächenänderung persistieren (geschlossene Architektur)

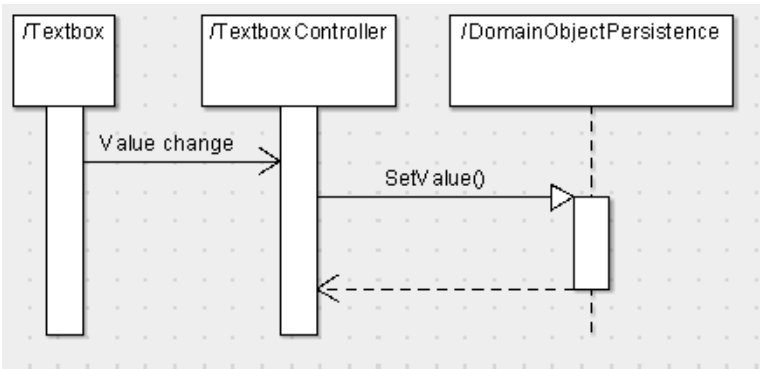

Abbildung 3.3: Sequenzdiagramm - Oberflächen Änderung persistieren mittels offener Architektur

Komplexe Softwaresysteme müssen in der Regel sehr lange am Markt bestehen. Bei Entwicklungszeiten von über 5 Jahren ist es nicht wirtschaftlich, wenn sich das System

	Offen	Geschlossen
Vorteil	hoher Freiheitsgrad	einfachere Logik
Nachteil	höhere Komplexität	"nicht" lösbare Probleme

Tabelle 3.1: Vor- und Nachteile von offener und geschlossener Architektur

nur ein Jahr am Markt hält. Das stellt die Architekten vor Probleme. Vor allem bei der offenen Architektur kann es zu einer übergenerischen Implementierung[17] kommen und das System zu komplex werden. Ist dies der Fall wird es für Entwickler schwer das System noch zu verstehen und es kommt zu einem erhöhten Fehleraufkommen und die Testbarkeit des Systems verschlechtert sich. Bei einer geschlossenen Architektur können leicht Use-Cases übersehen werden, die anschließend nur mit sehr viel Mühe ins System einzubringen sind.
Ob die Entscheidung des Architekten eingehalten werden, lässt sich leider kaum automatisiert[18] überprüfen. Daher ist der Dialog mit den Softwareentwicklern und ständige Code-Reviews, die Pflicht der Softwarearchitekten.

3.2 Unterscheidung nach der Programmiersprache

Es ist möglich .NET Assemblies in verschiedenen Programmiersprachen zu entwickeln und aus anderen Programmiersprachen aufzurufen. Dies wird ermöglicht durch die Common Language Specification (CLS) und das Common Type System (CTS). Microsoft liefert mit C#, VB.NET, C++/CLI, F#, JScript.NET und J# schon eine Reihe von .NET basierten Sprachen. Die Liste an zusätzlichen .NET Sprachen ist um ein Vielfaches länger Boo, Component Pascal, IronPython, IronRuby, IronLisp, #Smaltalk und viele andere mehr.
Ein Assembly wird in einer Sprache geschrieben und das generierte Ergebnis, kann aus anderen Sprachen verwendet werden. Da jeder Programmierer andere Vorlieben hat und der Einsatz von mehreren Sprachen im Konfigurationsmanagement zu Mehraufwand führt, ist es die Aufgabe des Architekten zu entscheiden, welche Programmiersprachen für die Realisierung des Projektes freigegeben sind.

3.3 Unterscheidung nach dem Aufbau

Ein Softwaresystem anhand seines Aufbaues zu unterteilen ist nicht einfach. Software kann meist nicht nur einem Typen zugeordnet werden. Es ist für den Architekten aber wichtig die prinzipielle Einteilung mit den Vor- und Nachteilen zu kennen.

[17] Übergenerisch ist ein Softwaresystem, wenn es zu allen Seiten offen ist, damit jeder denkbare (und nicht abzusehende) Use Case erfüllt werden kann. Dies steigert die Komplexität enorm.

[18] Microsoft FxCop - ein Programm zu statischen Quelltext analyse von .NET Programmen bietet prinzipell die Möglichkeit eigene Regeln zu definieren. Dafür könnte man eine Regel schreiben, das Assembly 1 Namensräume aus Assembly 2 nicht verwenden darf.

3.3.1 Multithread

Ein Thread ist ein leichtgewichtiger Prozess. Threads wurden eingeführt, als man erkannt hat, dass ein Prozesswechsel recht zeitintensiv ist (es muss der komplette Adressraum gesichert werden). Threads ermöglichen paralleles Arbeiten mit kürzeren Latenzzeiten, im Vergleich zur Umsetzung derselben Aufgabe mit unterschiedlichen Prozessen. In einem Adressraum können mehrere Threads laufen, was die Kommunikation zwischen ihnen erleichtert. Um zu verhindern, dass mehrere Threads gleichzeitig denselben Speicherbereich lesen oder schreiben muss dieser mit Betriebssystem Mechanismen wie "kritische Bereiche" (Critical Sections) geschützt werden. Es ist nicht möglich, alle Aufgaben einer Anwendung zu parallelisieren. Einige Aufgaben sollten aber immer in einem eigenen Thread realisiert werden:

- Zeichnen der Oberfläche
- Langläufige Berechnungen
- Laden von Dateien sollte in einem anderen Thread geschehen

Bei Interoperabilität mit COM ist oft auch ein separater Thread unumgänglich. Viele COM Server müssen aus dem einem Thread im sogenannten MultiThreadApartment State aufgerufen werden. Der Hauptthread einer .NET-Anwendungen ist aber klassisch im SingleThreadApartment State (Attribut [STAThread] über der Main-Methode).
Da die Leistung pro CPU sinkt (Stromsparende Modelle), dafür die Anzahl der Cores innerhalb einer CPU zunimmt, bringt nur paralleles Arbeiten in Zukunft Performance Vorteile. Microsoft hat dies erkannt und in das .NET Framework 4.0 die Task Parallel Library (TPL) [19] integriert, um das parallele Programmieren zu vereinfachen. Mit dem .NET Framework 4.5 wird C# zwei neue Schlüsselwörter unterstützen (*async* und *await*)[20]. Diese sollen das parallele Programmieren vereinfachen.

3.3.2 Multiprozess

Definition nach Torsten Posch (aus [Posch, Seite 215]):

> "Wenn die einzelnen Bausteine der Architektur in separaten Prozessen laufen, ist jeder durch einen eigenen Adressbereich vor der Außenwelt geschützt."

Die Aufteilung des Programmes auf verschiedene Prozesse bietet einem viele Möglichkeiten, wie einen größeren Adressraum, Parallelität oder den ersten Schritt in Rich-

[19] Eine Beschreibung der Task Parallel Library finden sie unter `http://msdn.microsoft.com/de-de/library/dd460717.aspx`

[20] Ein Überblick über die neuen Schlüsselwörter ist unter `http://community.devexpress.com/blogs/markmiller/archive/2011/01/05/mads-torgersen-on-the-future-of-c-and-the-new-async-keyword.aspx` zusehen.

tung "Verteilte Anwendung". Das große Problem stellt neben der sinnvollen Verteilung von Aufgaben die Synchronisation der Prozesse dar. Es gibt verschiedene Varianten zur Realisierung einer Inter-Prozess-Kommunikation (IPC). Die bekanntesten sind Pipe, Socket, SharedMemory, Remoting, Windows Communication Foundation oder Datenaustausch-Dateien. Die richtige Variante für IPC zu wählen und die Austausch Schnittstellen zu definieren ist komplex. Die große Gefahr ist es, dass Prozess A nur auf Daten von Prozess B wartet.

3.3.3 Client-Server

Die wohl bekannteste Anwendung von Client-Server Architekturen ist das World Wide Web (www). Über den Unified Ressource Locator (URL) wird ein Webserver angesprochen. Dieser verarbeitet die Anfrage und sendet das Ergebnis an den Webclient (Browser) zurück. Danach kann der Client das Ergebnis anzeigen.
Die Technologie COM basiert auch auf dieser Architektur. Es gibt einen COM-Server, der einen Dienst anbietet und von COM-Clients genutzt werden kann. Eine in COM realisierte Anwendung ist meistens mehrfach Server und mehrfach Client zur selben Zeit. Oft besitzt nur der Client eine Oberfläche und ein Server kann gleichzeitig mit mehreren Clients arbeiten. Dies führt zu einer klaren Einteilung von Zuständigkeiten.
Ein Webserver generiert mittels serverseitiger Programmiersprachen (PHP, ASP.NET) eine HTML Seite. Anschließend wird die HTML-Seite über TCP/IP an den Webclient übermittelt. Der Webclient ist dafür zuständig, aus dem HTML Dokument eine Oberfläche zu erzeugen und diese anzuzeigen. Dies erfolgt meist unter Verwendung von Cascading Style Sheets und clientseitigen Scriptsprachen wie JavaScript. In der Architektur muss festgelegt und verankert werden:

- Welche Aufgabe hat der Client/der Server?
- Wie kommunizieren Client und Server?
- Welche Prozesse gibt es (z.B. ein Clientprozess und ein Serverprozess oder selber Prozess)?
- Kann ein Client auch Server sein?
- Wie arbeitet der Client wenn der Server nicht mehr erreichbar ist?

Client-Server Architekturen helfen bei einer einer sauberen Trennung zwischen Anzeige und Logik. Der Overhead bei der Kommunikation zwischen Server und Client wirkt sich natürlich nachteilig bei den Laufzeiten aus. Auch ist die Fehlersuche in Zusammenspiel von Client und Server nicht unproblematisch.

3.3.4 Verteilt

In einer Client-Server-Architektur arbeiten ein oder mehrere Clients mit einem Server zusammen. Heutzutage ist es in Wirklichkeit aber nicht mehr nur ein Server. Die Aufgaben des Servers werden über Load Balancing auf unterschiedliche Server verteilt, wobei alle Server dieselbe Datenbasis zum Arbeiten verwenden. Zudem kann die Server-Software auf vielen unterschiedlichen Rechnern (Cloud-Computing) laufen.
Bei verteilten Anwendungen existiert klassischerweise ein Master-Server, der die Aufgaben an die Slave-Server weiterverteilt. Wenn der Master-Server ausfällt, übernimmt einer der Slave-Server diese Aufgabe. Je mehr Rechner in verteilten Anwendungen zum Einsatz kommen, desto besser ist das System skalierbar und umso eher kann das erzielte Ergebnis erreicht werden.
Beim SETI@Home-Projekt kann jeder seinen privaten Rechner zur Verfügung stellen, um bei der Suche nach außerirdischem Leben zu helfen. Dazu werden auf den Rechner Daten eines Radioteleskops übertragen, welche nach bestimmten Mustern durchsucht werden.
Mit verteilten Architekturen erreicht man neben großen Rechen-/Speicherkapazitäten auch eine gute Ausfallsicherheit. Bei Quelltextverwaltungssystemen gibt es zentrale Lösungen wie Rational ClearCase oder Microsoft TFS und die verteilten (dezentralen) Lösungen wie git oder svn. Bei den dezentralen Systemen ist es kein Problem, wenn das Main Repository ausfällt, da jeder Nutzer eine lokale Kopie der Quelltextdateien besitzt und somit weiter arbeiten kann.

3.4 Nach der Oberflächentechnologie

In .NET ist es sehr einfach Oberflächen zu erstellen. Jedoch gibt es eine Vielzahl von Oberflächentechnologien. Die Vor- und Nachteile der einzelnen Technologie zu kennen und die Entscheidung, welche verwendet wird, ist eine typische Architekturaufgabe.

3.4.1 Windows Forms

Der Namensraum System.Windows.Forms (kurz WinForms) bietet seit .NET 1.0 Zugriff auf die Klassen der Klassenbibliothek, welche sich zum Erstellen von grafischen Oberflächen eignen. Abbildung 3.4 zeigt ein Beispiel. Das zentrale Objekt bei dieser Oberflächentechnologie ist das Graphics Object. Es bietet die Möglichkeit Linien, Rechtecke, Texte, Polygone und vieles andere mehr zu zeichnen. Graphics verwendet zum Zeichnen die Schnittstellen[21] *Graphical Device Interface (GDI)* oder *Graphical Device Interface+ (GDI+)* von Windows.
Die Verwendung von GDI hat den Vorteil, dass das erstellte Programm ohne Zusatz-

[21] .NET 1.x basierte komplett auf GDI+, ab .NET 2.0 basieren zum Teil Steuerelemente wieder auf dem klassischen GDI.

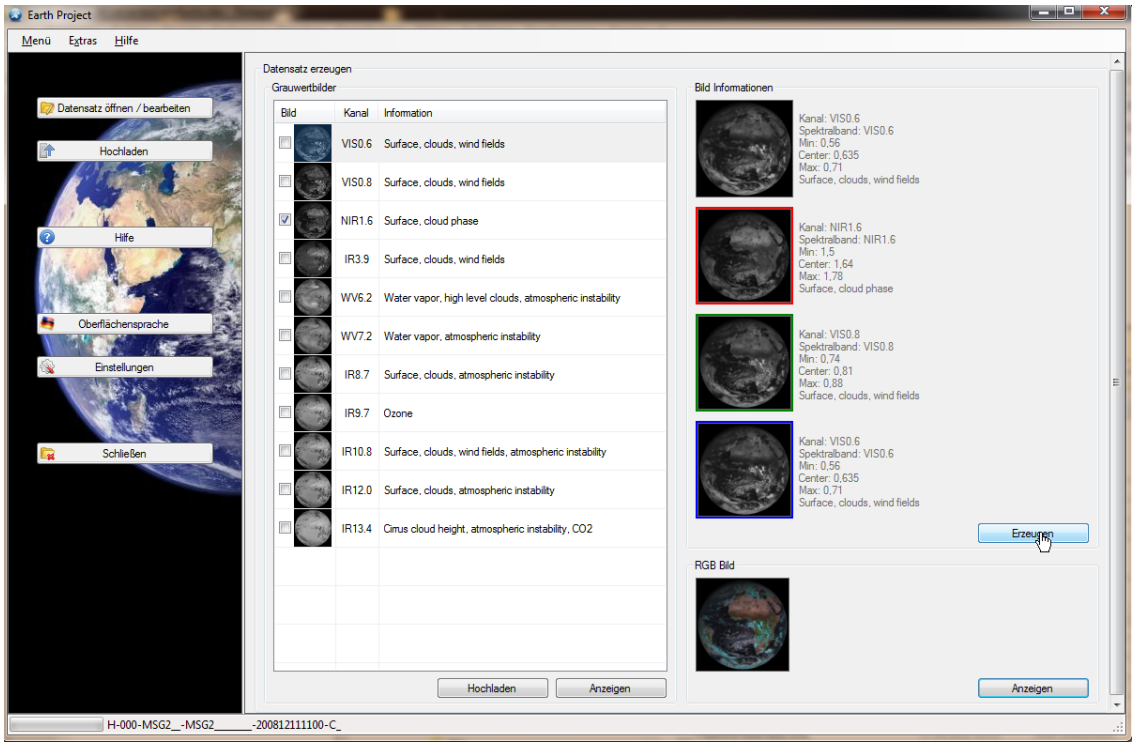

Abbildung 3.4: Anwendung mit einer Windows Forms Oberfläche

software über Remote Desktop angezeigt werden kann. Der Nachteil ist, dass GDI+ von der CPU und nicht von der GPU gerendert wird. Der WinForms Namensraum beinhaltet eine ziemlich umfangreiche Control-Bibliothek. Für nicht enthaltende Oberflächenelemente (wie Charts) gibt es jede Menge freie und kommerzielle Control-Bibliotheken. Abbildung 3.5 zeigt eine Anwendung mit einfachem Koordinatensystem zum plotten von Graphen. Der Quelltext des Oberflächenelementes befindet sich im Anhang unter Graph-Plotter WinForms Usercontrol.
Für das Entwickeln solcher Oberflächenelemente werden keine Kenntnisse in Vektorgrafiken notwendig. Wichtig ist nur, dass die obere linke Ecke des Zeichenbereiches die Koordinaten x=0 und y=0 besitzt und die untere rechte Ecke die Koordinaten x=Breite (Width) und y=Höhe (Height).
Die textuelle Beschreibung für Layout und Inhalt von Anwendungen (vergleiche Formulare bei Visual Basic 6.0) unterstützt die WinForms nicht, da sie komplett über Quelltext beschrieben werden. Mittels Reflektion und einigen Aufwand ist es jedoch möglich die Layout-Funktionalität eigenständig zu implementieren. Die WinForms unterstützen folgende Funktionalitäten (aus [Schwichtenberg, Seite 704]):

- "Positionierung von Steuerelementen auf einem Formular in einem Steuerelementbaum
- Dynamische Veränderung des Steuerelementbaums
- Nutzung von Steuerelementen von Drittanbietern
- Erstellung eigener Steuerelemente

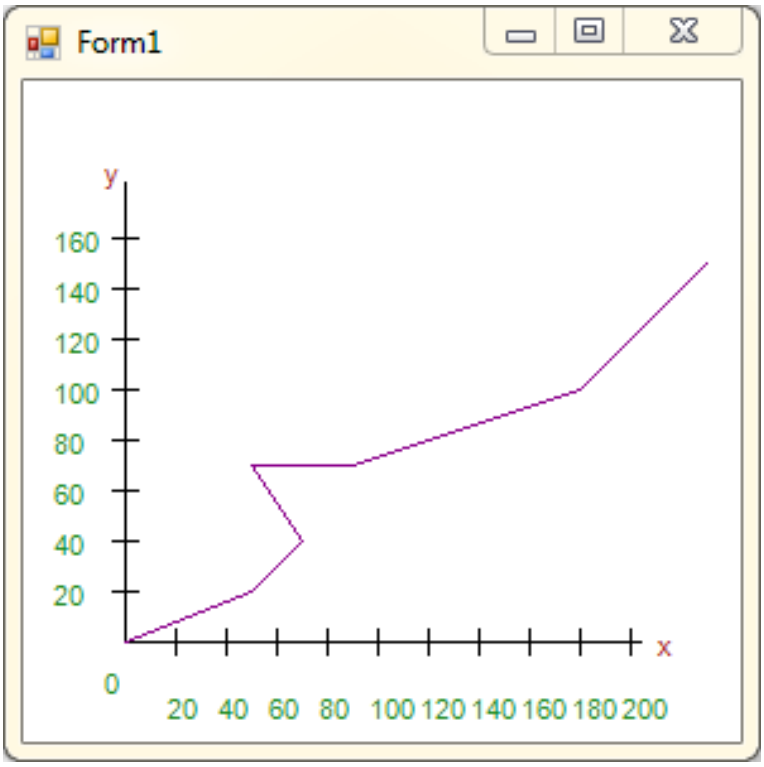

Abbildung 3.5: WinForms-Anwendung mit selbst gebautem Graphen-Plotter

- Vererbung von Steuerelementen und Fenstern
- Datenbindung
- Automatische Mehrsprachigkeit von Fenstern (Lokalisierung)
- Fenster mit Kinderfenstern (Multi-document-Interface-Anwendungen)
- Drag & Drop
- Zwischenablage
- Zeichnen auf der Formularfläche und in Steuerelementen
- Drucken
- Deployment von Windows Forms-Awendungen über XCopy-Deployment, No-Touch-Deployment oder Click-Once-Deployment
- Hosting von Windows Forms-Steuerelementen im Internet Explorer"

3.4.2 Windows Presentation Foundation (WPF)

Alle Elemente im Namensraum System.Windows (Ausnahme: System.Windows.Forms Namensräume) beschreiben Klassen der WPF. Es ist eine durchgängige Bibliothek zum erstellen von Oberflächen. Im Gegensatz zu WinForms bietet WPF mehr Möglichkeiten, u.a.:

- 3D-Oberflächen
- im Browser gehostete Applikationen

- Dokumente
- Videos

WPF ist neu implementiert und keine Erweiterung zu WinForms, sondern dessen Ersatz. Wie lange Microsoft WinForms weiter unterstützen wird, kann momentan nicht gesagt werden.
Um eine schrittweise Migration von WinForms nach WPF zu ermöglich, empfiehlt Microsoft erst den äußeren Rahmen einer Applikation in WPF zu schreiben und mittels *Element Host* und *Windows Forms Host* die alten Oberflächen zu hosten. Nach und nach sollten diese dann durch WPF-Implementierungen ersetzt werden.
Das Visual Studio 2010 ist die erste große Anwendung von Microsoft, deren Oberfläche komplett in WPF umgesetzt wurde. Abbildung 3.6 zeigt einen einfachen WPF - WYSIWYG Blog Editor, mit Rechtschreibprüfung und Undo/Redo-Funktionalität. Lose

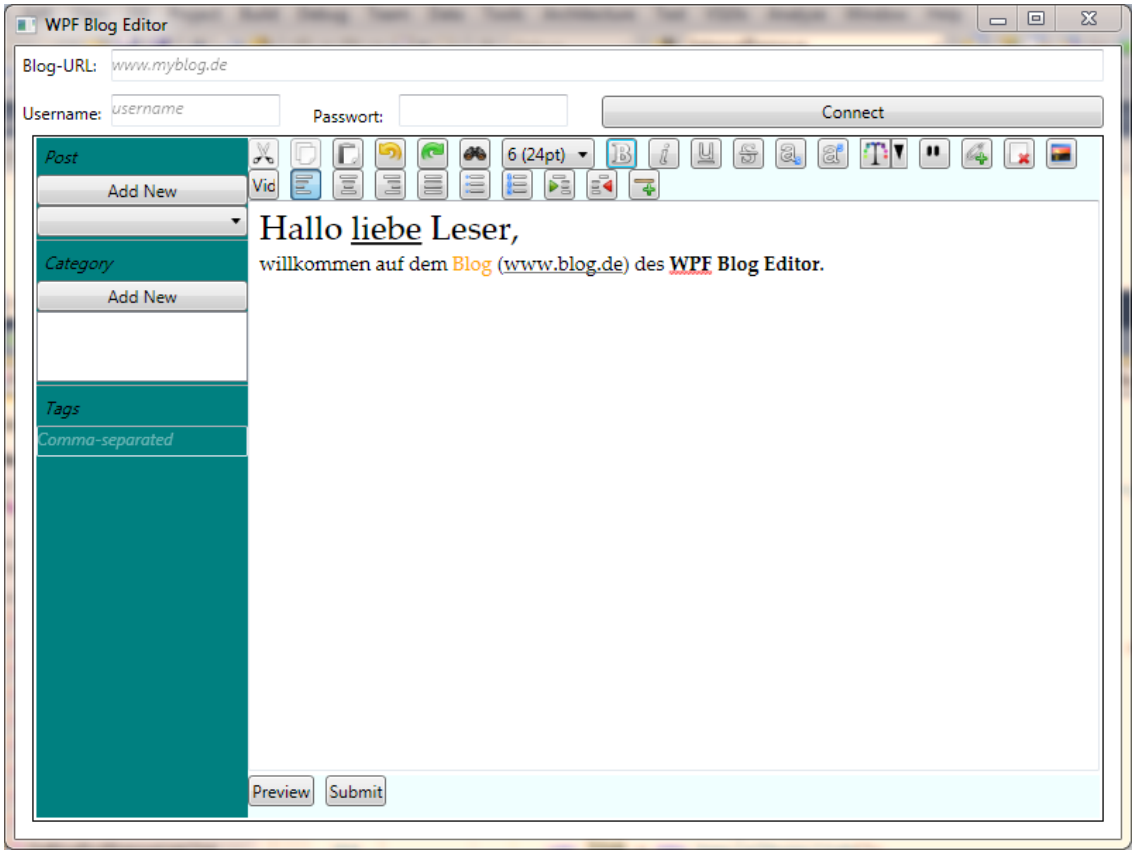

Abbildung 3.6: Mit WPF implementierter WYSIWYG-Blog-Editor

Kopplung zwischen Oberflächenelementen wird durch Kommandos (Commands) realisiert. Damit nach einem Klick auf einen Button anschließend in einem Textfeld alles unterstrichen dargestellt wird. Dafür ist nur folgender XAML-Code[22] innerhalb eines Fensters notwendig:

[22] Für das Beispiel wurde die Extensible Application Markup Language gewählt. Um den deklarative Ansatz zu zeigen, es wäre auch möglich gewesen, die Oberfläche in C# Code zusammenzubauen.

```
<ToggleButton Name="BlogPostUnderlineButton"
  Height="22"
  Width="22"
  ToolTip="Underline"
  Command = "EditingCommands.ToggleUnderline"
  CommandTarget="{Binding␣ElementName=RichTextBoxPost}">
    <Image Source="pics/format-text-underline.png"  />
</ToggleButton>

 <RichTextBox Name="RichTextBoxPost"
  SpellCheck.IsEnabled="True" >
    <RichTextBox.Resources>
      <Style TargetType="{x:Type␣Paragraph}">
        <Setter Property="Margin" Value="0"/>
      </Style>
    </RichTextBox.Resources>
    <FlowDocument FontFamily="Palatino␣Linotype" FontSize="14"/>
 </RichTextBox>
```

Listing 3.1: Definition einer Oberfläche über den deklarativen XAML-Ansatz.

Damit das Beispiel auch Undo/Redo-fähig ist, benötigt es nur zwei weitere Buttons, welche die Commands ApplicationCommands.Undo bzw ApplicationCommands.Redo an die RichTextBox senden. WPF unterstützt folgende Funktionalitäten (aus [Schwichtenberg, Seite 704]):

- "Oberflächen können wahlweise in Programmcode (wie in Windows Forms) oder - bevorzugt - durch die XML-basierte Extensible Application Markup Language (XAML) definiert werden.
- Trennung von Code und Gestaltung ist möglich. Dadurch können Benutzeroberflächen von Anwendungen zukünftig einfacher von ausgebildeten Gestaltern erstellt werden. Bisher werden Benutzeroberflächen oft von Entwicklern erstellt, denen es an einer Ausbildung im Bereich Ästhetik und Benutzerfreundlichkeit fehlt.
- WPF-Oberflächen laufen als eigenständige Windows-Fenster, im Fenster eines Browsers oder in speziellen Viewern
- WPF-Anwendungen können als clientseitige Browser-Anwendung laufen, verwenden dann jedoch kein HTML, sondern setzen WPF auf dem Client voraus. Vollständiges WPF als XML Browser Applicaion (XBAP) setzt ein vollständiges .NET Framework auf dem Client voraus. [...]
- Die Darstellung erfolgt intern über DirectX
- Die Anzeige ist vektorbasiert und bietet daher eine gute Darstellung unabhängig von der Größe des Anzeigegerätes. (Um eine gute Dar-

stellung auf allen Bildschirmgrößen und -auflösungen zu erzielen, verwendet WPF als Einheit sogenannte geräteunabhängige Pixel, die dem 96. Teil eines Inch entsprechen.)
- Steuerelemente können sich der Größe ihres Inhalts anpassen.
- Unterstützung für 2-D- und 3-D-Grafiken
- Unterstützung für Navigationsanwendungen (Hyperlinks und Vor/zurück im Stil einer Webanwendung)
- Unterstützung für an die Fenstergröße anpassbare Anordnung der visuellen Elemente
- Unterstützung für fest und flexibel umbrechende Dokumente (ohne Programmcode)
- Deklarative Datenbindung für alle Eigenschaften
- Abspielen von Videos
- Transformationen und Animationen von Oberflächen
- Definition von wieder verwendbaren Formatvorlagen (Styles)
- Definition von eigenen Steuerelementen (User Controls)
- Installation über XCopy-Deployment, klassische Installationsroutinen (inklusive Microsoft Windows Installer - MSI) oder automatischer Download (einschließlich Click-Once-Deployment)[...]
- Drag & Drop
- Verwendung der Zwischenablage in WPF
- Interoperabilität zu Windows Forms, COM / ActiveX, MFC, DirectX"

WPF unterstützt Microsoft UI Automation (UIA), welche es bespielsweise ermöglicht, zugreifbare Schnittstellen für Blindenlesegeräte zu erstellen, sich aber auch für automatisiertes UI Testen eignet.
Die Nachteile für den Einsatz von WPF sind:

- Remote Desktop ist nur möglich, wenn der Client auch WPF installiert hat.
- WPF ist ein ganz neues UI-Konzept. Wissen aus Arbeiten mit WinForms ist eher hinderlich als vorteilhaft. Vorwissen in den Bereichen ASP.NET, GDI, DirectX und Dynamic HTML (DHTML) ist hilfreich. Der Einarbeitungsaufwand ist erheblich.
- Lokalisierung von Oberflächen ist komplizierter als bei WinForms.[23]
- Ein häufig genannter Performance-Tipp[24] heißt "Reduce number of visuals". Dies zeigt das WPF mit dem Rendern von anspruchsvollen Oberflächen möglicherweise Probleme hat.

[23] Auf Codeplex wurde ein 65 Seitiges Whitepaper veröffentlich welches sich mit der Lokalisierung von WPF Anwendungen beschäftigt. `http://wpflocalization.codeplex.com/`

[24] Die besten Performance-Tips für WPF sind unter `http://blogs.msdn.com/b/jgoldb/archive/2007/10/10/improving-wpf-applications-startup-time.aspx` und `http://www.wpftutorial.NET/10PerformanceTips.html` zusammengefasst.

- Der WPF-Editor in Visual Studio wird immer besser, ist aber noch unvollständig. Microsoft Expression Blend wird als zusätzliches Programm benötigt, um professionelle Oberflächen zu entwerfen.
- Die Anzahl von Steuerelementen ist um ein Vielfaches kleiner als bei WinForms.
- Geräte ohne leistungsfähige Grafik-Hardware rendern die Oberfläche fast komplett auf der CPU, was die WPF-Anwendung im Gegensatz zu einer vergleichbaren WinForms Anwendung sehr viel langsamer erscheinen lässt.

Abbildung 3.7 zeigt eine Anwendung mit einfachem Koordinatensystem zum plotten von Graphen. Der Quelltext des Oberflächenelementes befindet sich im Anhang unter Graph-Plotter WPF Usercontrol.

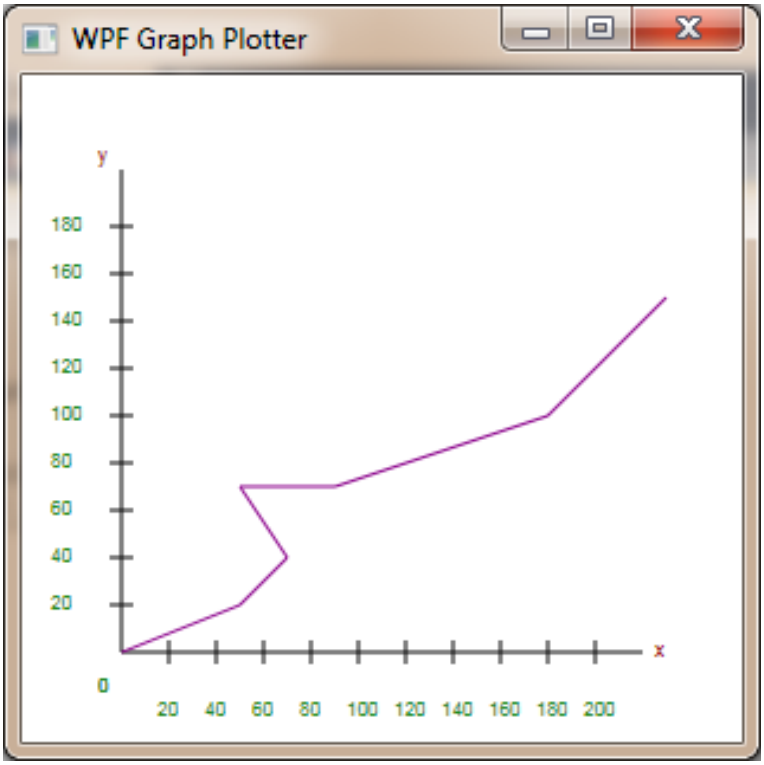

Abbildung 3.7: WPF-Anwendung mit selbst gebauten Graphen-Plotter

3.4.3 METRO

Mit Windows 8 liefert Microsoft ein Betriebssystem, welches sowohl für Desktop-PCs als auch für Tablets geeignet ist. Dies wird zum einen dadurch erreicht, dass Windows 8 ARM-Prozessoren unterstützt und zum anderen, dass es jetzt zwei Desktops gibt. Den klassischen Desktop und den Tablet Desktop (im folgenden METRO-Desktop genannt), siehe Abbildung 3.8.
Programme für den METRO-Desktop (auch METRO-Style-Apps oder kürzer METRO-Apps genannt) unterscheiden sich von klassischen Desktop-Programmen in einigen

Abbildung 3.8: Windows8 Tablet-Desktop (METRO-Desktop)

Punkten.

- Die Oberflächen-Technologien sind XAML oder HTML5 mit CSS.
- Ein Methodenaufruf, der mehr als 50ms dauert, muss asynchron erfolgen.
- Eine gestartete METRO-App lebt solange, bis das Betriebssystem keinen freien Speicher mehr hat, dann wird diese terminiert.
- In einer Manifest-Datei müssen alle Rechte, welche die App benötigt, hinterlegt sein. Sollte eine App Methoden aufrufen, wozu sie keine Rechte besitzt, wie das Ansteuern einer Webkamera, so wird dies vom Betriebssystem unterbunden.
- Die Logik von METRO-Apps kann wahlweise in C, C++, C#, VB.NET oder JavaScript implementiert werden.
- Zur Implementation steht nicht das .NET Framework oder die Win32 Schnittstellen des Betriebsystems bereit, sondern es gibt eine neue Betriebssystem Bibliothek, die WinRT (windows runtime). Sie weist folgende Eigenschaften auf:

  - komplett asynchron und nativ programmiert
  - Unterstützung von Zugriff und Modifikation von XAML-Dateien
  - Enthält zur Zeit ca. 1800 Klassen

- Da dieser Desktop für den Einsatz auf Geräten mit Touchscreen gedacht ist, ist eine veränderte Usability notwendig. Kontextmenüs, die über Rechtsklick angesteuert werden, sind nicht angebracht.
- METRO-Apps laufen immer im Vollbild Modus (full screen).

In Abbildung 3.9 zeigt einen Überblick der Windows 8 Plattform. Obwohl die WinRT

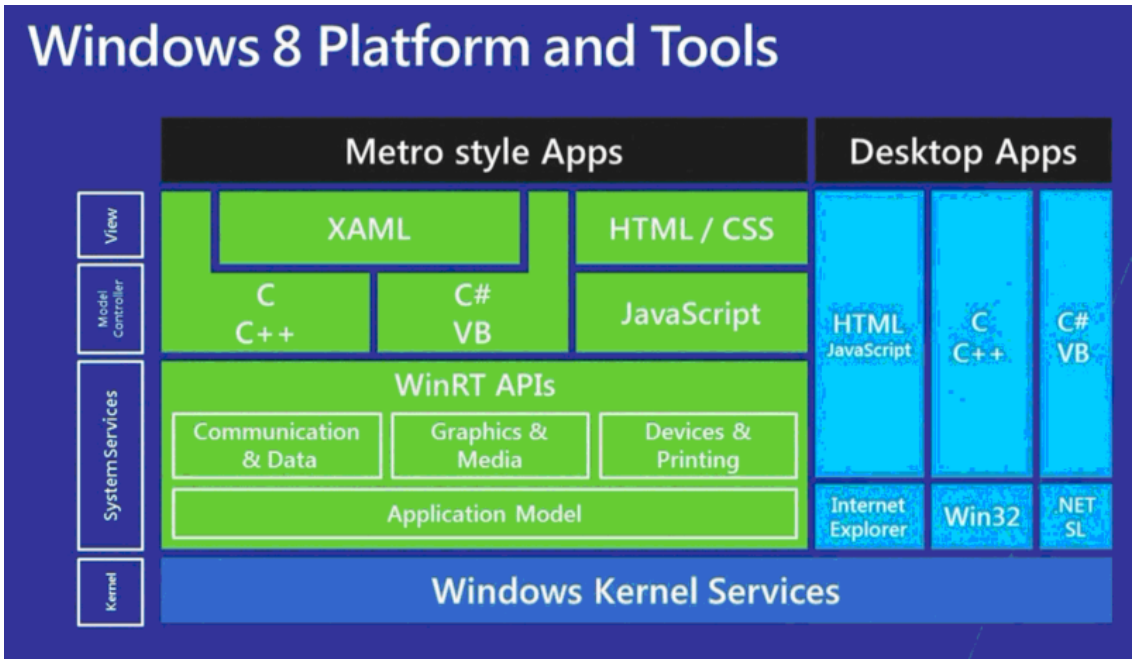

Abbildung 3.9: Überblick die Plattformen und Werkzeuge in Windows 8 (Bildquelle: Microsoft)

XAML unterstützt und XAML standardisiert ist, ist es nicht ohne weiteres möglich WPF-XAML oder Silverlight-XAML in einer METRO-App zu verwenden. Dasselbe Problem hat man schon, wenn man versucht ein XAML zu erstellen, das sowohl von WPF als auch von Silverlight verwendet werden soll. Man könnte sagen, mit METRO ist ein dritter XAML-Dialekt hinzugekommen. Dies ist ganz klar ein Nachteil von METRO. Weitere Nachteile sind, dass es die für METRO benötigte WinRT nur für Windows 8 geben soll, und die Tablet-Bedienung es schwer macht, Oberflächen für komplexe Software zu entwerfen.
Ein Vorteil, dass man mit bekannten Programmiersprachen und XAML, Apps für Tablets entwickeln kann, ist nicht von der Hand zu weisen. Sehr wahrscheinlich macht es keinen Sinn, die Oberfläche für ein komplexes Softwaresystem auf Basis von METRO zu entwickeln. Aber es sind andere Verwendungsmöglichkeiten denkbar:

- Statistik App für Präsentationen
- Bedienen und Beobachten App für Analysezwecke
- Service-App, sollte eine Server-Software ein Problem erkennen, könnte eine Service-App mit Informationen versorgt werden.

Solche zusätzlichen Möglichkeiten könnten einen Mehrwert gegenüber Konkurrenzprogrammen bedeuten. Wichtig ist, dass die Architektur der Software entsprechende Schnittstellen für den Zugriff auf Daten einplant. Abbildung 3.10 zeigt einen einfachen Feedreader als METRO-App. Die Umsetzung des Graph-Plotters als METRO-App (sie-

Abbildung 3.10: Als METRO app implementierter Feedreader

he Abbildung 3.11) verwendet andere Farben, als das bei den bisherigen Beispielen der Fall war. Dies liegt daran, das die statische Klasse Colors der WinRT wesentlich weniger vordefinierte Farben bietet und darauf verzichtet wurde, diese zu erweitern. Der komplette Quelltext findet sich im Anhang unter Graph-Plotter METRO Usercontrol. Das vorliegende Kapitel behandelt nur die XAML-Umsetzung von METRO-Apps und nicht die Möglichkeiten von HTML 5 mit CSS und JavaScript. Die Verwendung von XAML bietet Synergie-Effekte. Auch wenn WPF-XAML anders ist als METRO-XAML findet man sich schnell zurecht.

3.4.4 Silverlight

Silverlight wurde von Microsoft als Konkurrent zu Adobe Flash entwickelt. Es ist als Browser-Plugin für viele Browser und Betriebssysteme verfügbar (auch auf mobilen Ge-

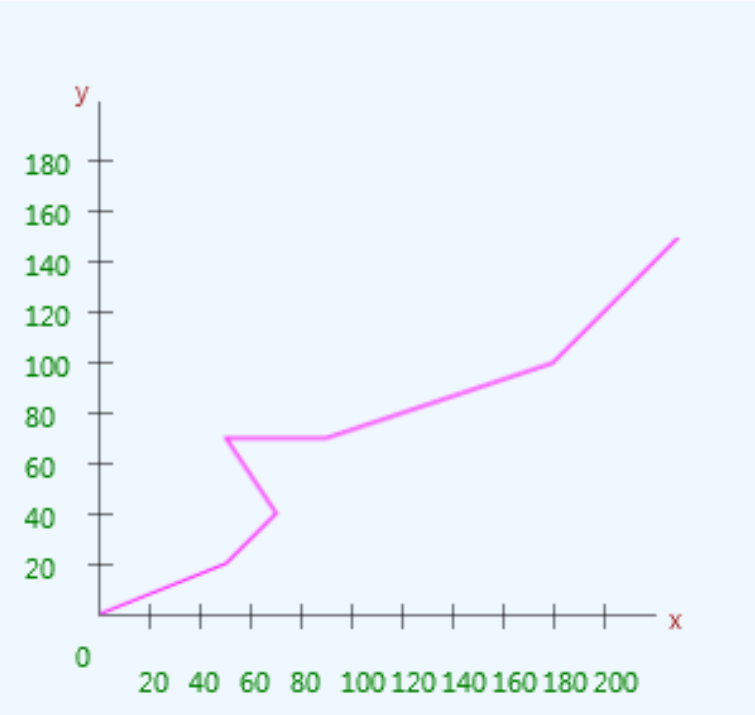

Abbildung 3.11: Als METRO app implementierter Graph-Plotter

räten wie dem Windows Phone7 oder einem Android Tablet über Moonlight[25]). Es arbeitet dabei clientseitig. Die Oberflächen werden (wie bei WPF oder METRO) in XAML beschrieben oder programmatisch festgelegt. Für die Realisierung von Anwendungen stehen eine Reihe von Funktionen zur Verfügung.

- Darstellung von 2D-Grafiken, 3D-Grafiken sowie Animationen
- Video-Wiedergabe
- Drag & Drop
- Funktionen einer eingeschränkten CLR-Base Class Library[26]
    - Generics
    - Reflection
    - Regex
    - Threading
    - Dateisystem Zugriffe
    - LINQ
    - (De)Serialierung
    - XML-Dokument

[25] Moonlight ist eine freie Open-Source Version um Silverlight-Anwendungen auf Unix, Linux und MacOS laufen zu lassen. Es wird vom Mono-Team entwickelt, weitere Informationen sind zu finden unter `http://www.mono-project.com/Moonlight`

[26] Seit Silverlight 2.0 ist es möglich, Funktionalitäten in CLR Sprachen wie C# oder VB zu entwickeln. Vorher waren nur Scriptsprachen (JavaScript, Phyton und Ruby) möglich. Damit konnte der Vorteil einer streng typisierten Sprache mit Compiler-Unterstützung und IntelliSense in der IDE genutzt werden.

    - HTML-Dokument
    - Interoperabilität mit nicht verwalteten Code über P/Invoke
- Out of Browser[27]
- Webcam und Mikrofon-Unterstützung
- Zusammenarbeit mit WCF-RIA Services (Windows Communication Foundation-Rich Internet Application)
- Multi-Touch
- Sandbox mit Isolated Storages[28]
- 64-Bit und vollständige GPU Unterstützung [29]

Abbildung 3.12 zeigt die Architektur von Silverlight. Dort ist deutlich zu erkennen, das Silverlight Komponenten von WPF verwendet werden. Der frühere Name von Silverlight war WPF/E (Windows Presentation Foundation Everywhere) und deutet darauf hin, dass Silverlight als eingeschränktes WPF fürs Web gedacht war. Seitdem WPF-Anwendungen direkt im Web gehostet werden können und Silverlight-Anwendungen außerhalb eines Browsers lauffähig sind, entwickelte sich Silverlight eigenständig weiter.
Silverlight eignet sich gut dafür, um ASP.NET Webanwendungen grafisch und funktionstechnisch zu erweitern, aber auch um Geschäftsanwendungen umzusetzen. Die Umsetzung eines Model-View-ViewModels ist laut Dr.Joachim Fuchs[30] in Silverlight 5 einfacher zu realisieren als in WPF. Der Aufwand für eine Umschulung von WPF ist minimal, für jemanden der nur Erfahrung mit Oberflächen in Windows Forms besitzt ist der Aufwand aber enorm (ähnlich wie bei der Umschulung von WinForms zu WPF). Hinzu kommen neue Hindernisse bei der Entwicklung, sei es dass man für jeden Browser extra Zwischenspeicherung (Caching) deaktivieren muss oder dass es nicht auf Anhieb klappt, im Webbrowser zu debuggen[31]. Abbildung 3.13 zeigt einen Graph-Plotter als Silverlight-Anwendung im Webbrowser. Die Implementierungsdetails des Graph-Plotters befinden sich im Kapitel A.1.8.
Wenn man im Internet nach Silverlight allgemein oder speziell nach Silverlight 6 re-

[27] Seit Silverlight 3.0 ist es möglich Anwendungen lokal zu speichern und auch außerhalb des Browsers auszuführen.

[28] Seit Silverlight 4.0 laufen die Anwendungen in einer Sandbox und haben (ohne explizite Zustimmung vom Benutzer) keine Rechte auf dem Dateisystem zu arbeiten. Um trotzdem mit verschiedenen Dateien (wie Hintergrundbilder) arbeiten zu können gibt es Isolated Storages. Wird eine Anwendung als Out of Browser betrieben, werden die Beschränkungen der Sandbox zum Teil aufgehoben. Somit wird auch das Laden von COM-Komponenten möglich.

[29] Mit Silverlight 5.0, was Ende 2011 freigegeben werden soll, nutzt Silverlight 64-Bit-Prozessoren besser und kann auf der Grafikkarte rendern.

[30] Dr. Joachim Fuchs ist Mitautor des bekannten Buches "Webanwendungen mit ASP.NET 3.5 und Ajax Crashkurs" und beschäftigt sich seit Begin auch mit Silverlight. In einem Artikel auf Heise Developer beschreibt er seine Erfahrungen mit Silverlight 5 `http://www.heise.de/developer/artikel/Silverlight-5-in-der-Praxis-1380712.html`

[31] Selbst mit dem neuen Visual Studio 2011 inkl. SP1, klappt es oft nicht ohne Workarounds Fehler zu suchen. Die Standard Lösungen erklärt Alex Norcliffe in seinem Blog `http://boxbinary.com/2010/04/debugging-silverlight-in-visual-studio-breakpoints-not-being-hit/`.

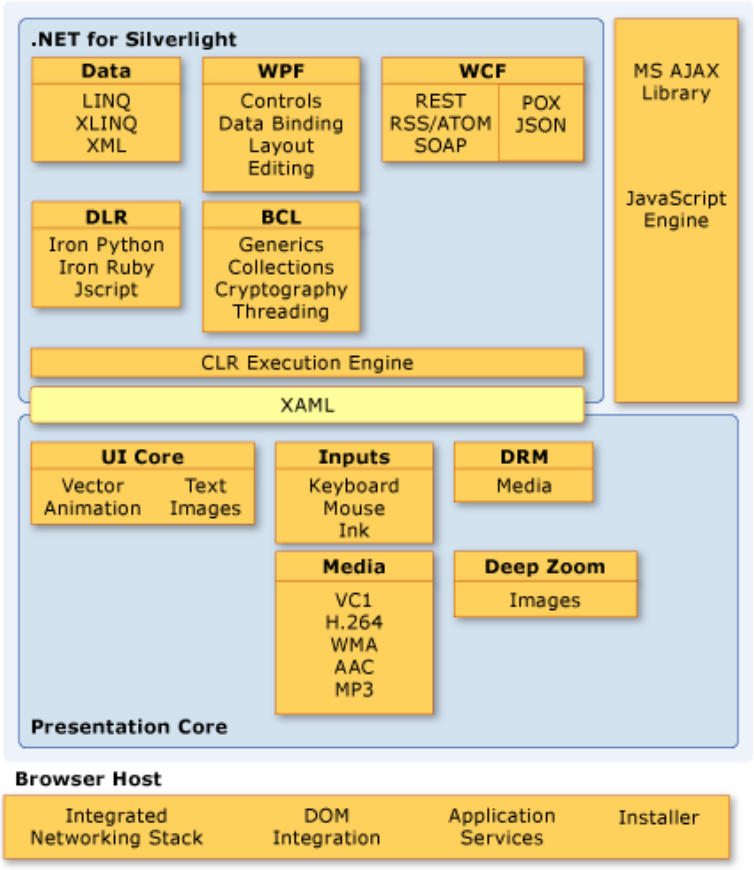

Abbildung 3.12: Architektur von Silverlight, Bildquelle: Microsoft

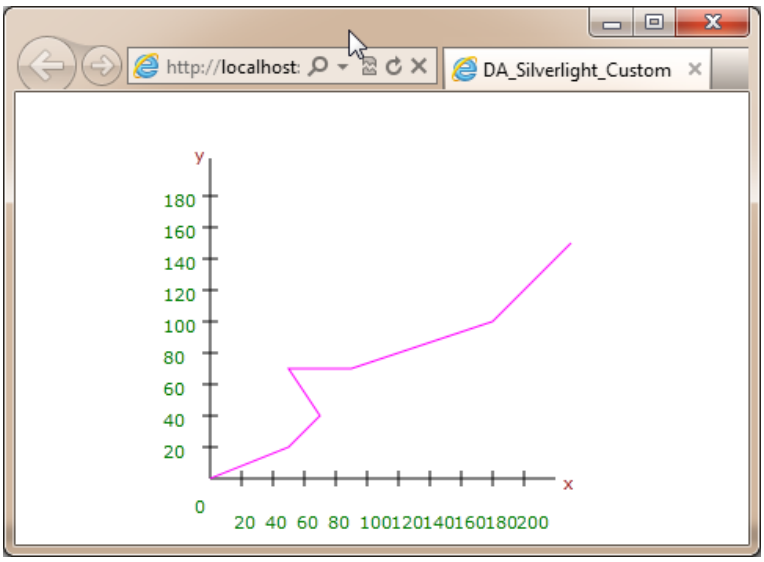

Abbildung 3.13: Silverlight Anwendung mit selbst gebautem Graph-Plotter

cherchiert findet man öfter die Aussage "Silverlight ist tot"[32]. Diese Aussage basiert auf

[32] In ihrem Blog gibt das Silverlight Entwickler-Team eine Übersicht über "Standardbasiertes Web, Plugins und Silverlight" `http://blogs.msdn.com/b/silverlight/archive/2011/04/04/standards-based-web-plug-ins-and-silverlight.aspx`

mehreren Faktoren:

- Plugin-basierte Webtechnologien wie Adobe Air, Adobe Flash oder Microsoft Silverlight, werden nicht vom Apples IOS und somit IPhone/IPad unterstützt.
- HTML 5 ist nach Meinung vieler die einzige Lösung für Cross-Plattform Web-Applikationen.
- Microsoft hat kein weiteres Major Release (das wäre 6.0) angekündigt.
- Laut Aussage diverser Webforen arbeitet ein Großteil der Silverlight Entwicklung jetzt bei Microsofts METRO-Team.

Silverlight wird laut Meinung der Community nicht mehr weiterentwickelt, allerdings bietet die aktuelle Version auch viele Funktionalitäten und ist auch für ältere Browser und Betriebssysteme verfügbar. Das kommende Windows 8 wird laut Aleš Holeček Silverlight auf jeden Fall noch voll unterstützen (BUILD-Konferenz 2011 aus dem Vortrag "Platform for Metro style apps").

> "All your investments in Win32, .NET and Silverlight will be preserved. You will be able to run those apps on Windows 8."

4 Favorisierter Ansatz

Frederick Brooks schrieb schon 1986 (aus [Brooks, Seite 177]):

> "*No Silver Bullet - Essence and Accident in Software Engineering*
> There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity."

Dieser Grundsatz hat sich bis heute bewahrheitet. Es gibt keine Lösung, welche in jedem Szenario funktioniert, daher muss man auch den favorisierten Ansatz im Zusammenhang sehen. Der Kontext dieser Diplomarbeit liegt in der Entwicklung von Desktop-Anwendungen im Windows-Umfeld, welche sehr große Datenmengen verarbeiten müssen. Um das große Umfeld weiter einzuschränken, geht es im Rahmen dieser Diplomarbeit um Anwendungen, die viele Daten ohne besonders aufwändige Grafik anzeigen müssen. Für CAD-Programme, Browser, Spiele, und andere Programme müssten alle Themen separat betrachtet und neu gewichtet werden.

4.1 Allgemein

Die Entscheidung ob eine offene oder geschlossene Architektur zu bevorzugen ist, kann nur für konkrete Szenarien getroffen werden. Allgemein lässt sich jedoch sagen, dass eine offene Architektur einen höheren Freiheitsgrad bietet und somit für sehr langläufig Projekte besser geeignet ist. Der Ansatz einer geschlossenen Architektur bietet dafür eine höhere Sicherheit.
Nachfolgend ein konkretes Beispiel:
Gegeben sei ein Netzwerk-Managmentsystem in dem alle Switche, Router, Server und Clients in einem Teilnetz dargestellt werden. Dabei können die einzelnen Teilnetze als Ring, Bus oder Sterntopologie angezeigt werden. In einer Sicht auf ein Teilnetz ist es möglich, die Teilnehmer zu bedienen und zu beobachten.
Dem Anwender wird der Status des Gerätes in der Oberfläche, mittels verschiedener Farben angezeigt. Wenn das Gerät in Ordnung ist, sieht ist der Status grün andernfalls rot, zum Beispiel in dem Fall, wenn ein Gerät nicht erreichbar ist. Die Informationen über den Status der Teilnehmer werden von Diensten geliefert, die die Daten in einer Persistenzschicht abspeichern. Das Programm besitzt für alle Teilnehmer am Netz eine Geschäftslogik, die unter anderem doppelte IP-Adressen erkennt und ein Securitymodell realisiert.
Nun möchte der Anwender eine tabellarische Anzeige aller Teilnehmer des gesamten Netzwerkes mit Anzeige des Status und der entsprechenden IP-Adresse. Bei der Implementierung der Anforderung kommt es immer wieder zu Abstürzen. Der Grund dafür

liegt in der großen Anzahl an Geschäftslogik-Objekten, die zur Anzeige geladen werden müssen (OutOfMemory-Ausnahmen).
Betrachtet man diese Anforderung nicht bereits beim Entwurf, führt dies meistens zu einer Destabilisierung des Gesamtsystems.
Bei einer geschlossenen Architektur kann man versuchen, die Teilnehmer in Typen einzuteilen und die Geschäftslogik typenbasiert zu laden. Dahingegen kann die Oberflächenschicht bei einer offenen Architektur die Daten direkt aus der Persistenz beziehen und diese zur Anzeige bringen. Die neue Anforderung kann umgesetzt werden, ohne Quelltext zu duplizieren und ohne bestehende Implementierungen der Geschäftslogik anzupassen.
In diesem konkreten Beispiel würde sich eine offene Architektur besser eignen. Ist es aber beim Netzwerk-Managementsystem sehr wichtig, dass es aufgrund von Programmierfehlern oder äußeren Einflüssen nicht zu inkonsistenten in der Datenhaltung kommt, wäre eine geschlossene Architektur zu bevorzugen.

4.2 Architekturaufbau

Für eine Desktop-Anwendung im Windows-Umfeld eignet sich im Allgemeine eine Multithreaded-Anwendung mit nur einem Prozess. Wenn sie in der Anwendung nicht viel parallelisieren lässt, dienen die Threads nur dazu, die Oberfläche von der Verarbeitung der Daten zu entkoppeln. Wenn viele Daten verarbeitet werden müssen, ist zusätzlich ein zweiter Prozess sinnvoll. In diesem Prozess arbeitet die Persistenzschicht. Wichtig ist, dass die Inter-Process-Communication-Schnittstelle, möglichst viele Daten in einer Aktion von einem Prozess in den anderen verschiebt. Synchronisation von Prozessen oder Threads kosten immer viel Laufzeit, daher sollte nicht jede Anfrage eines Datums sofort zu einen Prozesswechsel führen.
Die Anwendung mit einer verteilten Architektur zu implementieren sollte nur vorgesehen werden, wenn es Anforderungen an die Software gibt, die dieses erfordern. Verteilte Anwendungen sind sehr vielseitig aber auch komplizierter zu implementieren.
Multithread-Mulitprozess in einer Client-Server-Architektur ist möglich. Wenn die Software sich in eine COM-Umgebung einfügen soll, ist die Implementierung von COM-Servern und COM-Clients nicht zu umgehen. Eine Empfehlung, ob die Anwendung intern auch von der Client-Server-Architektur Gebrauch machen soll, kann nicht so einfach beantwortet werden. Im Kontext von Desktop-Anwendungen ist der Client-Server Ansatz eine mögliche Realisierungsform der Dienstorientierung. Sollte die Anwendung viele Funktionalitäten in Diensten anbieten können, dann empfiehlt sich eine Umsetzung mittels Client-Server-Architektur (und dem entsprechenen Entwurfsmuster[33]).

[33] Das Standardwerk von Entwurfsmustern für verteiltes Arbeiten(unter das auch der Client-Server Ansatz fällt) ist Frank Buschman, Kevlin Henny, Douglas C. Schmidt *Pattern-Oriented Software Architecture - A Pattern Language for distributed Computing* Volume 4 der POSA-Serie, ISBN 978-0-470-05902-9

4.3 Oberflächentechnologie

Die Entscheidung der Oberflächentechnologie ist im .NET-Umfeld zur Zeit sehr schwierig. Im Kontext einer Desktop-Anwendung, die viele Daten anzeigen soll aber keine besonderen grafischen Anforderungen erfüllen muss ist WinForms zu empfehlen.

- WinForms sind einfach zu erweitern
- fast alle .NET Entwickler haben Erfahrungen in dieser Technologie
- die Fehlersuche ist einfacher als bei anderen Technologien
- durch die Verwendung von GDI/GDI+ sind die Oberflächen klassisch aber dafür ordentlich gerendert

WPF oder Silverlight als Technologie empfiehlt sich trotz des nicht geringen Einarbeitungsaufwandes, wenn

- zu der Desktop-Anwendung auch eine als Web-Browser nutzbare Variante existieren muss;
- aufwendige 3D-Grafiken benötigt werden;
- Animationen unterstützt werden sollen;
- die Oberfläche von einem getrennten Designer-Team entwickelt werden soll.

Gibt es die Anforderung einer Tablet-Variante, kommt zur Zeit nur Silverlight in Frage, da Silverlight auf Android, dem klassischen Windows-Desktop, aber auch auf dem zukünftigen Windows 8 lauffähig ist. Die Umsetzung einer Anwendung als METRO-App ist nur sinnvoll, wenn Windows 8 als Zielplattform ausreichend ist.

5 Auswirkungen der Architektur im .NET-Umfeld

Dieses Kapitel dient dazu, Probleme, die eine Architektur in großen Softwaresystemen hat, zu beschreiben und wenn möglich Lösungen zu diskutieren.

5.1 Allgemeine Probleme in großen Projekten

In diesem Kapitel werden allgemeine Probleme von Softwarearchitekturen in großen Softwaresystemen besprochen. Wobei die Diskussion aus Sicht des .NET-Umfeldes beschrieben wird.

5.1.1 Schwergewichtige Objekte

Um eine strikte Schichtenarchitektur einzuhalten, ist es verpflichtend, dass eine untere Schicht keine Abhängigkeiten zu einer ihr überlagerten Schicht besitzt. Des Weiteren darf eine obere Schicht nur auf die Ihr direkt unterlagerte Schicht zugreifen. Es ist besonders zu empfehlen, dass eine Schicht niemals Annahmen über Implementierungsdetails einer anderen Schicht trifft, sondern sich lediglich auf den öffentlichen Kontrakt bezieht. Dafür ist es eventuell notwendig Quelltext zu schreiben, um Informationen von einer unterlagerten zu einer überlagerten Schicht durchzureichen. Dadurch ergibt sich, dass praktisch jede Schicht einen eigenen Wrapper/Container um die Objekte der unterlagerten Schicht erstellen muss. Jede Schicht erweitert die Funktionalitäten eines Objektes und bietet dadurch dem Softwareentwickler einen Mehrwert. Eine höhere Laufzeit ist durch die strikte Schichtenarchitektur bedingt, genauso wie ein erhöhter Speicherverbrauch. Wenn man die Größe von Objekten im Speicher betrachtet, dann gillt:

- Die Größe des Programmcodes ist unkritisch, da die CLR diesen nur einmal im Speicher hält. Dies gilt auch wenn es mehrere Instanzen einer Klasse gibt.
- Die instanzspezifische Daten (Felder, Events) sind ausschlaggebend für den Speicherverbrauch, da diese Daten pro Instanz abgelegt werden.

Daher lässt sich schlussfolgern, dass wenn die Wrapper der einzelnen Schichten keine Instanzdaten (wie Caches) besitzen, ist der Mehrverbrauch an Speicher zu vernachlässigen.
Dem erhöhten Speicherverbrauch kann entgegen gewirkt werden, wenn Caches kontextbezogenen eingesetzt werden. Somit können bestimmte Use-Cases bereits in höheren Schichten bearbeitet werden ohne den Zugriff auf unterlagerte Schichten. Dies beeinflusst die Laufzeit und den Speicherverbrauch positiv, da Objekte in unterlagerten

Schichten entladen bleiben können bzw. nicht geladen werden müssen. Insbesondere funktioniert dieses Konzept bei datenintensiven Projekten, da dort nicht alle Objekte im Speicher gehalten werden können, sondern ausgelagert werden müssen, zum Beispiel in eine SQL-Datenbank.
Die strikte Einhaltung des Schichtenmodells ist nur eines von vielen Beispielen, wie es zur Bildung von schwergewichtigen Objekten kommen kann. Schwergewichtig sind die Objekte, weil z.B. jede Schicht, die Objekte der ihr unterlagerten Schicht dekoriert (Dekorierer-Pattern) oder einen Wrapper darum erzeugt.
Dieses Vorgehen ist trotz der Nachteile, dass die Objekte einen erhöhten Speicherverbrauch und eine erhöhten Laufzeit haben, ideal. Zum Vergleich soll die Umsetzung der Funktionalitäten mittels Ableitung dienen. Jede Klasse leitet sich von dem Objekt der ihr unterlagerten Klasse ab und erweitert dieses. Dadurch erbt das Objekt alle Informationen des Vaterobjektes, das steht im Gegensatz zu dem allgemeinen Ziel, Details von unterlagerten Schichten zu verstecken. Das Testen von speziellen Funktionalitäten über Mocken, Optimierungen wie Lazy Loading oder dass entladen von unterlagerten Schichten und eine dynamische Bindung von Funktionalitäten ist nicht mehr möglich.
Vererbung sollte nur für die logische Strukturierung der Objekte verwendet werden. Eine Umsetzung konkreter Anforderungen sollte mittels Wrappern oder Dekoration erfolgen.
Das .NET Framework arbeitet mit Implicit Invocation, dies ermöglicht über Events, dass vom Framework gesteuert Anwendungscode ausgeführt wird. Doch *Events* halten die schwergewichtigen Objekte am Leben. Es ist nur eine Frage der Menge von Objekten, wann sich dies zum Problem entwickelt. Es muss auch die Frage geklärt werden wann sich die Logiken Registrieren und Deregistrieren. Im Falle der Registrierung wär die logische Antwort "Wenn die überlagerte Schicht den Auftrag erteilt ein Objekt anzulegen, wird eine einmalige Initialisierungs-Routine der Logiken durchlaufen". Diese Aussage ist aus mehreren Gründen sehr gefährlich:

- Sollte die Software irgendwann multiuserfähig werden, kann es sein, dass Objekte auch plötzlich in einer unterlagerten Schicht existieren, ohne dass dies explizit von einer oberen Schicht beauftragt wurde.
- Einmalig bedeutet hier pro Domänenobjekt, das heißt, es wird eventuell recht häufig aufgerufen.
- In welchen Geltungsbereich (scope) ist das "einmalig" gültig? Pro
    - aktueller Benutzer-Session?
    - Instanzierung des konkreten Objektes?
    - Laufzeit des Programmes?
    - Verwendung des Projektes?

Im Falle der Deregistrierung ist die Frage sehr schwer zu beantworten. Das Kapitel Lebenszyklus von Objekten sollte zeigen, dass es nicht ausreicht, davon auszugehen, dass eine Objekt-Wolke verschwindet, wenn die Oberfläche dass "oberste" Objekt loslässt. Es ist sehr wichtig den Schritt zum deregistrieren zu definieren, da sonst die Ob-

jekte nach dem einmaligen Laden wahrscheinlich sich immer im Speicher befinden. Eine saubere Umsetzung ist es, dass die Schichten-Objekte *IDisposable* implementieren und dabei allen Logiken die Möglichkeiten gegeben wird, sich überall zu deregistrieren. IDisposable ist eine Schnittstelle des .NET Frameworks, die dazu dient Ressourcen freizugeben, sobald sie nicht mehr benötigt werden. Das Freigeben von Ressourcen im Finalizer, geschieht erst, wenn der GC das Objekt abräumt. IDisposable hat die Vorteile, dass es in die Programmiersprache integriert ist (über das Schlüsselwort using) und es eine Tool-Unterstützung (Regeln für FxCop) gibt. Die FxCop Regeln zeigen einem Quelltextstellen, bei denen der Aufruf der Dispose Methode vergessen wurde.
Um das mehrfache Ausführen der Initialisierungs Methoden zu verhindern, könnte man die Registrierungen persistieren und die Logiken nur dann laden, wenn das Ereignis eingetreten ist für das sich die Logik interessiert und eine *Callback-Methode* aufrufen. Dieser Ansatz bedeutet bei kleineren Objekt-Mengen einen gewissen zeitlichen Mehraufwand würde aber Speicher sparen, wenn es viele Instanzdaten pro Logik gibt.
Um den Kerngedanken der Optimierung aufzugreifen, ist es auch denkbar, die komplette Registrierung komplett in *Metadateien* vorzunehmen. Dies ist allerdings nur sinnvoll möglich, wenn alle Objekte ausmodelliert sind. Dazu hier ein Beispiel: Die Schichten der Anwendung sollten nicht nur die Klasse Shape kennen, wenn an der Oberfläche in Wirklichkeit mit Triangle, Circle und Square gearbeitet wird. Es muss also eine typbasierte Klassifikation geben. Mit den definierten Typen ist es auch möglich einige Merkmale, die für alle Instanzen eines Typen gleich sind (z.B. Registrierungen) an den Typen zu hängen, anstatt diese Daten instanzgranular zu speichern. Letztlich bietet sich ein Typen-System unabhängig von einer Meta-Beschreibung zur Notifikation immer an.

5.1.2 Abhängigkeiten von Objekten

Die Realisierung bestimmter Anforderungen bedingt, dass Objekte untereinander abhängig sind. Objekt A muss vielleicht etwas machen, wenn Objekte B etwas getan hat. Es gibt viele verschiedene Abhängigkeiten, ob es nun Objekte aus Domänen sind (vgl. Kapitel 2.2.1), oder voneinander abhängige dynamisch ladbare Funktionalitäten (siehe Kapitel 2.2.2). Die zentrale Frage ist, wie sollen Abhängigkeiten untereinander aufgelöst werden. Events sind dafür geeignet; damit weiß der Sender nichts von dem Empfänger, und alle Empfänger werden am Leben gehalten. Es ist auch möglich, das Eventing so zu modifizieren, dass die Empfänger nicht im Speicher gehalten werden:

```
1 public class CustomWeakEvent
2 {
3   private IList<WeakReference> m_Events;
4   [MethodImpl(MethodImplOptions.Synchronized)]
5   public CustomWeakEvent()
6   {
7     m_Events = new List<WeakReference>();
8   }
```

```
9   public event EventHandler CustomEvent
10  {
11    add
12    {
13      lock (m_Events)
14      {
15        m_Events.Add(new WeakReference(value));
16      }
17    }
18    remove
19    {
20      WorkOnEvents(
21        (handler, list, index) =>
22        {
23          bool shouldBreak = false;
24          if (handler.Target == value.Target
25            && handler.Method == value.Method)
26          {
27            m_Events.RemoveAt(index);
28            shouldBreak = true;
29          }
30          return shouldBreak;
31        });
32    }
33  }
34  public void RiseEvent(object sender, EventArgs e)
35  {
36    WorkOnEvents(
37        (handler, list, index) =>
38        {
39          handler.Invoke(sender ?? this,
40            e ?? EventArgs.Empty);
41          return false;
42        });
43  }
44  private void WorkOnEvents(Func<EventHandler,
45    IList<WeakReference>, int, bool> action)
46  {
47    lock (m_Events)
48    {
49      for (int i = 0; i < m_Events.Count; i++)
50      {
51        var handler = m_Events[i].Target
52          as EventHandler;
53        if (handler == null)
54        {
```

```
55              m_Events.RemoveAt(i);
56              i--;
57            }
58            else if (action.Invoke(handler, m_Events, i))
59            {
60              break;
61            }
62          }
63        }
64      }
65  }
```

Listing 5.1: Event Hantierung ohne angemeldete Objekte am Leben zu halten.

Für die Aktualisierung der Oberfläche oder eines Zwischenspeichers ist diese Lösung ideal. Es gibt sicherlich weitere Anwendungsfälle bei denen es gewünscht wird, dass die Empfänger die Notifikation nur empfangen, wenn sie am Leben sind. Doch in vielen Fällen trägt diese Lösung nicht, unter anderem, wenn die Empfänger persistente Daten ändern müssen. Persistente Datenänderungen sollten von einer Benutzeraktion ausgelöst werden und alle in dem selben UNDO-Schritt erfolgen. *Es gibt keine Lösung, die in jedem Fall optimal ist, um Abhängigkeiten aufzulösen. Man muss die verschiedenen Fälle differenziert betrachten.*

Erster Fall: Ein Container möchte alle Elemente benachrichtigen. Hierfür bietet es sich an, dass Elemente während der Instanzierung für bestimmte Informationen des Containers (wie eine Drucken-Aktion) anzumelden und ein eindeutiges Erkennungszeichen innerhalb des Containers (ID) und eine Methode hinterlegen. Wenn das Ereignis eintritt, ruft der Container die angemeldeten Elemente ins Leben und die Methode auf (Implicit Invocation).

Zweiter Fall: Ein Element darf bestimmte Funktionalitäten nicht anbieten, wenn ein anderes Element dies nicht unterstützt. Diese Änderung ist recht "lokal", dafür bietet es sich an, einen konkreten Vermittler (Mediator-Pattern) zu nutzen. Der Vermittler kennt vom zu informierenden Objekt ein eindeutiges Erkennungsmerkmal und die aufzurufenden Methode(n) und wird vom Sender aufgerufen, wenn das relevante Ereignis eintrifft.

Dritter Fall: Mehrere Elemente aus unterschiedlichen Containern beeinflussen sich gegenseitig. Hier geht es um sehr "globale" Abhängigkeiten. Eine Kombination aus Beobachter (Observer-Pattern) und Vermittler (Mediator-Pattern) bietet eine solide Lösung, um die Änderungen zu publizieren. Bei sehr komplexen Abhängigkeiten wird die Logik innerhalb des Vermittlers auch schnell unübersichtlich. Um die Komplexität zu reduzieren, empfiehlt es sich die einzelnen Aufgaben in geschachtelten Klassen (nested classes) zu implementieren und über eine Zuständigkeitskette (Chain of Responsibility-Pattern) die geeignete Aufgabe auszuwählen und auszuführen.

Gerade die Zuständigkeitskette mit den nested classes nutzt die Stärke des .NET Frameworks, kleine Objekte sehr schnell auf dem Heap anzulegen. Da es sich bei den

Aufgaben-Objekten um kurzlebige Objekte der Gen0 handelt, wird der Speicher auch schnell wieder freigegeben[34].

5.1.3 Navigation über das Modell

Dieser Punkt wird interessant, wenn Objekte viel Logik enthalten, die von anderen Objekten abhängig sind. Als Beispiel sollen Dreiecke verwendet werden. Jedes dritte Dreieck soll in einer grafischen Darstellung andersfarbig dargestellt werden. Ein Ansatz zur Realisierung ist eine zentrale Komponente, die jedes dritte Dreieck kennt und dafür sorgt, dass diese eine andere Farbe bekommt. Doch damit liegt das Wissen über diesen Spezialfall nicht beim Dreieck selbst, sondern bei einer anderen Komponente. Auch wird der Quelltext schnell unübersichtlich, wenn es für solche Spezialfälle immer neue Komponenten gibt (oder eine "Gott-Komponente", welche alles kennt). Der richtige Ansatz, ist dass die Dreiecke dieses Spezialwissen beinhalten sollten.
Dazu muss (wenn es keine speziellen Relationen zwischen den Dreiecken gibt) jedes

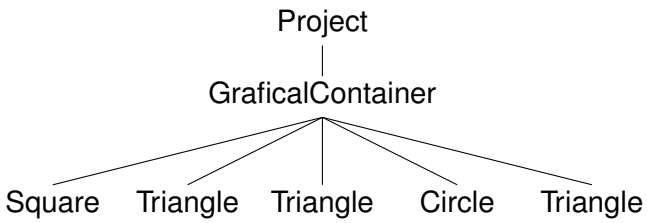

Abbildung 5.1: Beispielhafte Darstellung des Projektbaumes einer Anwendung

Dreieck zum Container und von dort über alle Elemente des Containers navigieren. Wie bereits im Kapitel Schwergewichtige Objekte beschrieben besteht das Problem, dass diese Objekte recht groß werden können und es dauert, bis alle geladen sind. In .NET-Anwendungen ist es auf Grund der Garbage Collection nicht möglich festzulegen, dass diese Objekte sofort wieder zerstört werden sollen. Durch das Initialisieren eines Dreieck-Objektes werden in Wirklichkeit ja pro Schicht mindestens ein neues Objekt angelegt, was gegen die Microsoft Regel "Avoid many Allocations"[35] verstößt.
Wenn viel navigiert werden muss, ist es aus Speicher- und Laufzeitsicht zu empfehlen, dass die unterste Schicht, welche die benötigten Informationen hat, direkt angesprochen wird um zum Beispiel die IDs der relevanten Objekte zu bekommen (SQL-Anfrage an die Datenbank). Mit der ID sollten dann die konkreten Objekte geladen werden. Durch dieses Vorgehen kann es aber wieder zu Problemen kommen, da nun

[34] Jeffrey Richter schreibt: "Die Leistungsmessungen von Microsoft haben ergeben, dass es weniger als eine Millisekunde dauert, eine Garbage Collection für die Generation 0 durchzuführen. Microsoft hat sich das Ziel gesetzt, die Garbage Collections so schnell zu machen, dass sie nicht länger als ein normaler Seitenfehler(Page Fault) dauern.

[35] Eine Beschreibung des Basiswissens über den Garbage Collector und Performance-Tipps ist zu finden unter `http://msdn.microsoft.com/en-us/library/ms973837.aspx#dotnetgcbasics_topic4`

eine obere Schicht etwas vom Aufbau einer unteren Schicht wissen muss. Es könnten zum Beispiel SQL-Befehle verwendet werden, die ein anderes DBMS (Datenbankmanagmentsystem) nicht unterstützt. Dadurch wäre ein Austausch des DBMS nicht mehr möglich.

5.1.4 Fehlersuche bei deskriptiven Ansatz

Es gibt keine fehlerfreie Software. Somit gehört es zu den täglichen Aufgaben der Softwareentwickler Fehler zu suchen und zu beheben. Bei der Verwendung von Architekturmustern wie dynamisch ladbare Funktionalitäten (siehe Kapitel Dynamisch ladbare Funktionalitäten) oder dem Model-View-ViewModel (Kapitel Model-View-ViewModel) bietet es sich an eine deskriptive Beschreibung zu verwenden. An dieser Stelle möchte ich einen Kollegen zitieren:

> "Jahrelang wurde an streng typisierten Programmiersprachen und unterstützenden Compilern entwickelt, nur um jetzt alle Typisierung über Bord zu werfen und das Binding über Strings zu realisieren."

Dieses Zitat beschreibt ein großes Problem, das durch die Verwendung einer deskriptiven Beschreibung entsteht. Durch dieses Verfahren geht im großen Umfang die Compiler-Unterstützung verloren, somit ist es nicht möglich, Fehler frühzeitig zu erkennen. Zur Fehlersuche kann man auch nicht mehr direkt an die Quelltextstelle des Aufrufes gehen, da Aufrufe im Allgemeinen über einen generischen Metaparser via Reflektion erfolgen. Im Gegensatz zum Ansatz ohne Metadateien gibt es nun mindestens 3 mögliche Fehlerquellen:

1. deskriptive Beschreibung
2. Parser für Beschreibungs-Dateien
3. eigentliche Implementierung der angesprochenen Funktionalität

Bei der Entwicklung von Oberflächen mit XAML kommt es öfter zu TypeLoadExceptions. Diese haben in den wenigsten Fällen mit der Einbindung der Namensräume zu tun. Aufgrund des Fehlers könnte man folgern, dass der Namensraum falsch eingebunden ist und der Typ wirklich nicht geladen werden konnte. Jedoch deutet der Fehler darauf, dass die deskriptive Beschreibung falsch ist. Beispielsweise können in einer Canvas-Umgebung nicht direkt mehrere Elemente gezeichnet werden. Es muss erst eine Canvas.Children Umgebung geöffnet werden. Für eigene Meta-Dateien ist zu empfehlen eine entsprechende Logging-Komponente zu erstellen, welche während der Interpretation der XML-Datei eine gut nachvollziehbare Logdatei erstellt. Die Logging-Komponente könnte folgendes XML ...

```
<Types>
  <Type name="Programm.PersonenVerzeichnis">
```

```
      <Assembly>C:\Program.Interface.dll</Assembly>
      <Ui name="Programm.PersonenVerzeichnisUiControl"
        assembly="C:\Debug\UiControls.dll"
        method="Init" />
    </Type>
</Types>
```

Listing 5.2: META-Beschreibung zu Aufbau einer Oberfläche mittels Reflektion.

... in einen gut nachvollziehbaren Log umwandeln:

```
27.11.2011 18:08:03.965 working on file "sample.xml"
27.11.2011 18:08:04.007 MD5: CF3E9CB701FE8153C359611BBC66A931
27.11.2011 18:08:04.021 Validating file
27.11.2011 18:08:04.888 --> no error
27.11.2011 18:08:04.100 Loading Types
27.11.2011 18:08:04.104 Using Assembly "C:\Program.Interface.dll"
27.11.2011 18:08:04.256 Loading type "Programm.PersonenVerzeichnis"
27.11.2011 18:08:04.504 Create Instance (ID:0815)
27.11.2011 18:08:04.583 --> no error
27.11.2011 18:08:04.983 Using Assembly "C:\Debug\UiControls.dll"
27.11.2011 18:08:05.042 Loading type "Programm.PersonenVerzeichnisUiControl"
27.11.2011 18:08:05.197 Create Instance (ID:4711)
27.11.2011 18:08:05.245 --> no error
27.11.2011 18:08:05.370 Call method "Init" at 4711 with 0815 as parameter
27.11.2011 18:08:05.712 --> fatalerror: InvalidCastException
```

Das im Beispiel verwendete XML ist sehr einfach und übersichtlich. In einer komplexen Anwendung wird dies wahrscheinlich nicht der Fall sein, sodass Logs bei der Fehlersuche sehr sinnvoll sind. Es passiert häufig, dass durch das Setup eine falsche Version der Metadatei ausgeliefert wird oder dass Kunden Meta-Daten bearbeiten, welche dazu nicht gedacht sind. Solche Fehler findet man ohne entsprechende Logs nicht. Wenn die Oberflächen der Anwendung hauptsächlich in XAML beschrieben sind, empfiehlt es sich mindestens einen Unit-Test zu schreiben, welcher die XAMLs auf Plausibilität testet:

- Sind die Namensräume korrekt?
- Sind die Assembly-Namen korrekt?
- Enthält Assembly a1 den Typen t1?
- Implementiert Type t2 das Interface i1?

Die Entscheidung, wie die Fehlersuche in Metadateien umzusetzen ist, muss von dem Architektur-Team getroffen werden, weil sie auch über den Einsatz der Metadateien

entschieden hat (und sei es nur die Entscheidung, die Oberfläche in WPF oder Silverlight zu entwickeln). Die Vorteile in der deklarativen Programmierung bestehen in der sauberen Schnittstelle und losen Kopplung von Komponenten. Durch den Einsatz von Dependency Injection oder Inversion of Control ist auch die Testbarkeit und Wiederverwendung gegeben. Die Vorteile müssen gegen die Nachteile (meist Laufzeitfehler, kaum Compiler-Unterstützung, schwierige Fehlersuche) abgewogen werden.

5.2 Spezielle Probleme im .NET-Umfeld

Dieses Kapitel behandelt spezielle Probleme der Softwarearchitektur, die es so mehr oder weniger nur im .NET-Umfeld gibt.

5.2.1 Interoperabilität

In diesem Kapitel geht es darum die Probleme aufzuzeigen die entstehen wenn eine verwaltete Anwendung native Programmteile verwenden möchte.
Die Verwendung von nativem Quelltext über **P/Invoke** hat folgende Nachteile:

- 10 bis 30 zusätzliche x86 Assembler-Anweisungen pro Aufruf der nativen Methode, sowie zusätzlicher Mehraufwand (overhead) wegen der Umwandlung (marshalling) der Datentypen, wobei bitgleiche Typen wie int und Int32 keinen Mehraufwand generieren. [36]
- Verhindert die Plattform-Unabhängigkeit, welche .NET-Anwendungen über Mono haben.
- Führt zu Problemen, wenn die Anwendung auf x64 anstatt auf x86 laufen soll. [37]
- Korrekte Implementierung in C# beinhaltet einige Fallstricke:
  - Pinnen von Objekten, welche im nativen Sourcecode benutzt werden sollen.
  - Mitteilung an den Garbage Collector, wie viel nicht verwalteter Speicher verwendet wird.
  - Marshalling-Attribute richtig setzen.
- Erschwerte Fehlersuche, zum einen kann man nicht einfach aus dem .NET-Quelltext in den nativen Quelltext debuggen. Zum anderen kann eine System.AccessViolationException ein Fehler in der C# Ansteuerung oder ein Speicher-Problem in den nativen Quelltext sein.[38]

Bei der Verwendung von COM ist folgendes zu beachten:

[36] Diese Aussagen sind eine Übersetzung aus dem Englischen aus dem Microsoft Developer Network `http://msdn.microsoft.com/en-us/library/ms235282.aspx`
[37] Siehe `http://msdn.microsoft.com/en-us/library/ms973190.aspx#64mig_topic4`
[38] Eran Sandler widmet sich in seinem Blog, diesem Thema `http://dotnetdebug.NET/2006/04/17/pinvoke-and-memory-related-issues/`

- MultiThreadApartment (MTA)-COM Server müssen aus einem seperaten Thread (mit Apartmentstate MTA) angesprochen werden.
- Bei Typ-Dateien mit Methoden, die Pointer als Parameter haben, kann die tlbimp.exe oft keine C# Zugriffsklassen generieren. In diesem Fall muss (am besten mit C++/CLI) die Zugriffsklasse selber implementiert werden. Dafür ist Wissen in der "normalen" COM-Programmierung notwendig.

Nachteile der Verwendung von COM sind der laufzeitliche Mehraufwand und die fehlende Plattformunabhängigkeit des Quelltextes. Ein potenzieller Nachteil ist das COM eine alte Technologie ist, d.h., wenn es keine erfahrenen COM-Programmierer im Entwicklerteam gibt, ist das Arbeiten mit COM fehleranfällig und die Fehleranalyse schwierig. Mit dem .NET Framework 4.0 hat Microsoft die Interoperabilität durch Einführung des neuen Schlüsselwortes *dynamic*[39] vereinfacht.

5.2.2 Obfuskation und Meta-Basierte Reflektion

Obfuskation[40] ist eine der wenigen Möglichkeiten um ein Reverse Engineering des IL-Code zu erschweren. Dazu werden im einfachsten Fall Meta-Informationen aus einem Assembly entfernt und Methoden umbenannt. Das folgende Listing zeigt den dekompilierten Quelltext ohne Obfuskation:

```
1  private static void Main(string[] args)
2  {
3    Program.PersistenceObject containerPersistence = new Program.
       PersistenceObject();
4    Program.BusinessLogicObject containerBusinessLogic = new Program.
       BusinessLogicObject(containerPersistence);
5    containerBusinessLogic.Init();
6    Program.PersistenceObject elementPersistence = new Program.
       PersistenceObject();
7    Program.BusinessLogicObject elementBusinessLogic = new Program.
       BusinessLogicObject(elementPersistence);
8    elementBusinessLogic.Init();
9    Program.PersistenceObject containerOfContainerPersistence = new
       Program.PersistenceObject();
10   Program.BusinessLogicObject containerOfcontainerBusinessLogic = new
       Program.BusinessLogicObject(containerOfContainerPersistence);
11   containerOfcontainerBusinessLogic.Init();
12   IList<Program.UIBase> objects = new List<Program.UIBase>(3);
13   objects.Add(new Program.ContainerUIObject(containerBusinessLogic));
14   objects.Add(new Program.ElementUIObject(elementBusinessLogic));
```

[39] ein Beispiel mit Erklärung wie dynamic bei COM Interop hilft ist im Microsoft Developer Network zu finden. `http://msdn.microsoft.com/en-us/library/dd264736.aspx`

[40] Eine interessanter Artikel des .NET-Magazin zum Thema Obfuskation befindet sich unter `http://msdn.microsoft.com/de-de/magazine/cc164058%28en-us%29.aspx`

```
15    objects.Add(new Program.ContainerOfContainerUIObject(
          containerOfcontainerBusinessLogic));
16    foreach (Program.UIBase ui in objects)
17    {
18      ui.Dowork();
19    }
20    for (int i = 0; i < objects.Count; i++)
21    {
22      objects[i].Dispose();
23    }
24  }
```

Listing 5.3: Dekompiliertes Programm ohne Obfuskation

Im folgenden Listing ist der dekompilierte Quelltext nach einer einfachen Obfuskation zu sehen:

```
1   private static void a(string[] A_0)
2   {
3     a.h a_ = new a.h();
4     a.d d = new a.d(a_);
5     d.a();
6     a.h a_2 = new a.h();
7     a.d d2 = new a.d(a_2);
8     d2.a();
9     a.h a_3 = new a.h();
10    a.d d3 = new a.d(a_3);
11    d3.a();
12    IList<a.c> list = new List<a.c>(3);
13    list.Add(new a.a(d));
14    list.Add(new a.g(d2));
15    list.Add(new a.b(d3));
16    foreach (a.c current in list)
17    {
18      current.b();
19    }
20    for (int i = 0; i < list.Count; i++)
21    {
22      list[i].Dispose();
23    }
24  }
```

Listing 5.4: Dekompiliertes Programm mit einfacher Obfuskation.

Der Kontrollfluss ist noch zu erkennen, aber das Lesen des Quelltextes wird wesentlich schwerer. Kommerzielle Obfuskation-Software geht noch weiter. Beispielsweise wird

der IL-Code mit Statements angereichert , die nie angesprungen werden (Deadcode Injection), oder For-Schleifen werden durch eine While-Schleifen mit Switch-Case-Anweisungen ersetzt.
Wer in .NET seine Assemblies vor Reverse Engineering schützen muss, muss bei der Architektur aufpassen. Nicht alle Frameworks für aspektorientiertes Programmieren unterstützen eine Zusammenarbeit mit Obfuskation. Hier gilt es, in einer Prototyping-Phase die richtigen Frameworks zu wählen. Werden Komponenten verwendet, die in Meta beschriebene Typ-Informationen auslesen und anschließend Instanzens mittels Reflektion erzeugen, so wird dies nach einer Obfuskation nicht mehr möglich sein. Dafür muss es im Generierprozess eine Komponente geben, welche die Mapping-Datei[41] der Obfuskations-Software einliest und mit Hilfe der Informationen die Metadateien entsprechend modifiziert.
Diese Lösung funktioniert nur, wenn alle Metadateien zum Generierungs-Zeitpunkt bekannt sind oder eine verschlüsselte (proprietäre) Datenstruktur mit den Informationen der Mapping-Datei mit im Programmverzeichnis liegt. Das ist notwendig wenn neue Metadateien hinzukommen können. Die neuen Metadateien beinhalten den "Klartext" der Methodenbezeichner. Für die Abbildung auf die internen Funktionen ist das Wissen aus der Datenstruktur notwendig.
Sicherheit von Software ist ein wichtiges Thema, dem im .Net-Umfeld zu wenig Beachtung geschenkt wird. Stack- oder heapbasierte Überläufe zur Codeausführung sind in .Net schwer zu realisieren. Solche ein Verfahren einzusetzen lohnt sich im Allgemeinen auch nicht, wenn der Quellcode des Programmes zeigt, wie man einen "Hook" implementiert. Wenn aspektorientierte Programmierung und/oder Reflektion in einem Softwaresystem verwenden werden soll, ist es unerlässlich, die Sicherheitsfrage während der Entwurfsphase zu klären. Erst kurz vor der Freigabe daran zu denken verhindert das Obfuskieren der Software.

5.2.3 Externe und interne Schnittstellen

Interfaces sind im .NET die bevorzugte Möglichkeit Schnittstellen zu definieren. Durch Implementierung eines Interfaces erfüllt eine Klasse einen Kontrakt. Nun kommt es auf die Art der Verwendung an, ob diese Schnittstellen sehr viele auch irrelevante Informationen enthalten (zu empfehlen bei Inter Process Communication) oder nur relevante Informationen enthalten (zu empfehlen bei Komponenten-Schnittstellen). In C++ ist es unüblich die abstrakte Klasse, welche sie als Parameter bekommen in eine bestimmte Klasse zu wandeln (Casten).
In .NET-Programmiersprachen sieht man öfter, das ein Interface in ein anderes Interface gecastet wird. Entstehen solche Quelltextstellen dann reicht das aktuelle Interface nicht und muss entweder erweitert werden oder die Möglichkeit bieten (über eine T GetS-

[41] In diesen Dateien schreibt eine Obfuskations-Software, welcher originale Name auf welchen Namen abgebildet wurde. Diese Dateien sind unersetzlich um Crash-Reports vom Kunden zu entschlüsseln oder obfuskierte-Software zu debuggen.

ervice<T>() Methode, oder eine an COM angelehntes IUnkown QueryInterface(Guid id)) zusätzliche Daten zu liefern. Die Implementierung muss aber auch immer damit arbeiten, dass *zusätzliche* Informationen eventuell nicht vorhanden sind. Eine unzureichende Schnittstelle einer Komponente ist ein Fehler in der Architektur. Da ein neuer Client, der die bestehende Komponente verwenden will und deswegen die geforderten Interfaces implementiert, erstmal nicht funktionieren wird. Um dieses Problem früh zeitig zu erkennen, müssen auch die Out-of-the-box-Funktionalitäten gegen die externen Schnittstellen implementiert werden.
Externe Schnittstellen enthalten oft auch eine Property vom Typ IDicitionary<string, object> oder NameObjectCollection (im folgenden "Argumente" genannt). Ob dies sinnvoll ist oder eine Entwurfsschwäche, hängt von der Verwendung ab. Es ist immer dann sinnvoll, wenn eine Komponente mehrere Clients informieren will und die Clients nutzen die Argumente als Zwischenspeicher von Ergebnissen oder auch zur Synchronisation untereinander nutzen. Es ist nicht sinnvoll, wenn durch Einträge in den Argumenten die Steuerung der Komponente verändert wird. In diesem Fall ist die Schnittstelle nicht ausreichend und hätte erweitert werden müssen, sonst wird die Compilerunterstützung wieder aufgegeben.
Wenn man in die Blackbox einer Komponente hineinschaut, so sind dort die internen Schnittstellen ordentlich entworfen. Es wird mit Strategien (Strategy-Pattern), Schablonenmethoden (Template-Method-Pattern) und Fabriken (Abstract-Factory-Pattern) gearbeitet, doch an der externen Schnittstelle ist das nicht ersichtlich. Oft liegt dies daran, dass Architekten und Entwickler sich die Möglichkeiten offen halten wollen größere Umbauten an den internen Mechanismen zu ermöglichen. Bei einer Offenlegung der Schnittstellen ist dies nicht mehr möglich. Wenn es aber die Wiederverwendbarkeit der Komponente steigert oder die interne Logik vereinfacht, dass auch außerhalb der Komponente zusätzliche Strategien implementiert werden können, sollte die internen Schnittstellen veröffentlicht werden.

5.2.4 Reflektion und Quelltextgenerierung

Reflektion (reflection) und Quelltextgenerierung bieten einem Architekten viele Freiheiten beim Design der Software. Beim Einsatz in großen Softwaresystemen kann der Einsatz dafür sorgen, dass das Programm an anderer Stelle abstürzt. Reflection und Quelltextgenerierung arbeiten beide viel mit Strings. Solange diese direkt verwendet werden, ist das unproblematisch. Bei häufiger Modifikation der Strings wird der Speicher unnötig befüllt (siehe Kapitel Zeichenketten). Strings können auch groß sein. Wenn z.B. eine ganze Quelltextdatei eingelesen wird, um sie den CodeDom.Compiler zu übergeben, ist es möglich, dadurch den LOH zu fragmentieren.
Quelltextgenerierung ist unproblematisch, solange man Dateien nicht erneut übersetzen muss. Die Generierung erzeugt im Speicher des Prozesses ein Assembly. Assemblies können nur entladen werden, wenn eine AppDomain entladen oder der Prozess ter-

miniert[42] wird. Das bedeutet, um den Speicher nicht unnötig zu füllen, müssen beim Arbeiten mit Codegenerierung immer AppDomains verwendet werden.
Wenn Probleme mit dem Speicherverbrauch existieren, in der Applikation aber nicht auf Quelltextgenerierung verzichten kann, sollte überprüft werden, ob die Generierung nicht in einem separaten Prozess erfolgen kann. Dessen Ausgabe-Assembly im Dateisystem gespeichert wird und aus der Hauptapplikation via reflection verwendet wird.

[42] Das MSDN-Magazin beschreibt in einem Artikel wie die Quelltextgenerierung zum Speicherloch (memory leak) führt. `http://msdn.microsoft.com/en-us/magazine/cc163491.aspx#S4`

6 Zusammenfassung

6.1 Darstellung der wesentlichen Ergebnisse

Abbildung 6.1 zeigt in schwarz die problematischen Eigenarten des .NET Framworks und farbig mögliche Lösungsansätze. Die nachfolgende Tabelle dient als Hilfestellung

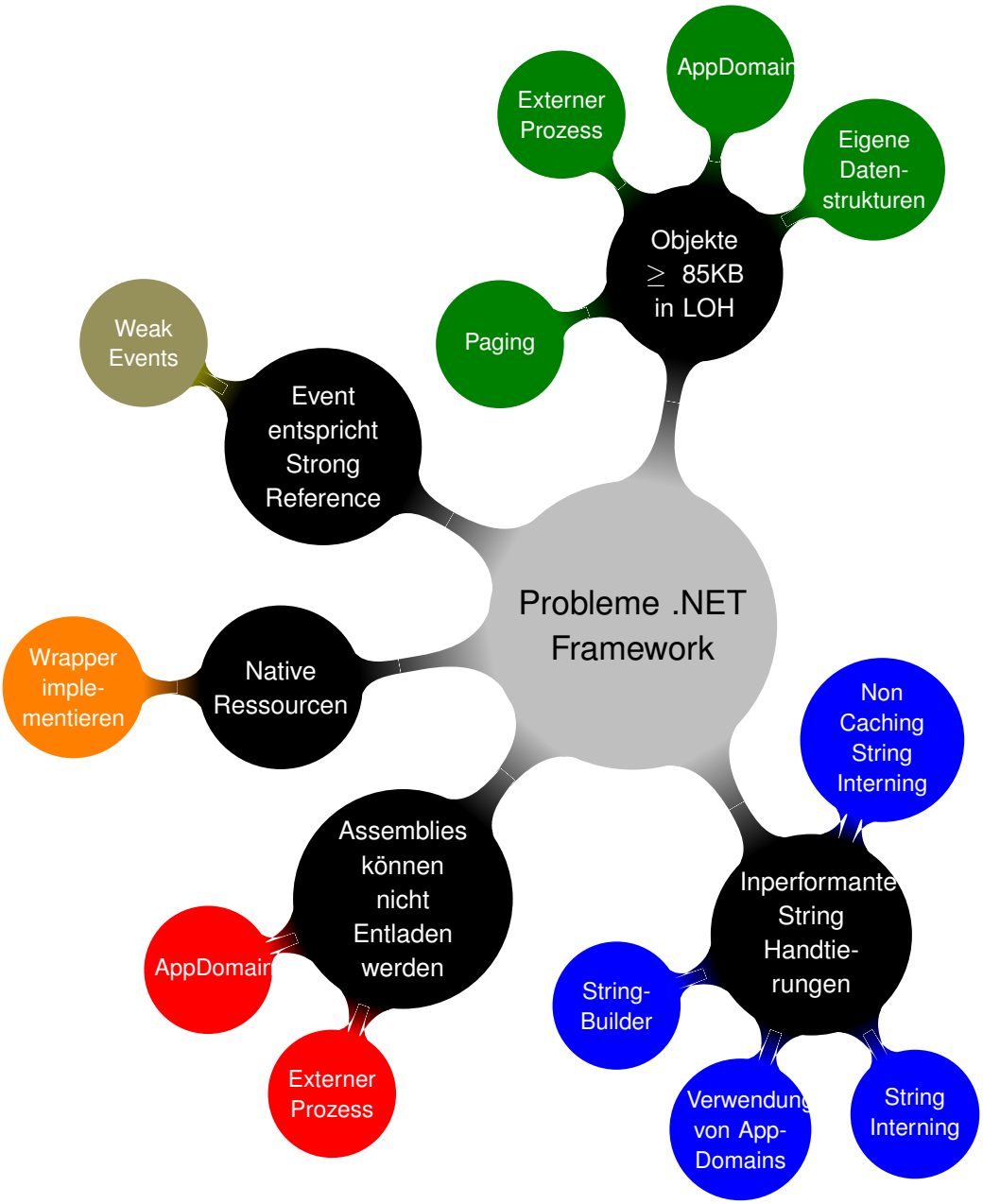

Abbildung 6.1: Übersicht der Probleme des .NET Frameworks mit Lösungsideen

für Softwarearchitekten und soll bei der Erstellung einer passenden Softwarearchitekten unterstützten.

Prinzip	Regeln	Beispiel
Entscheidungen müssen ganzheitlich betrachtet werden.	Die Umsetzung von funktionalen und nichtfunktionalen Anforderungen fügt in der Regel zusätzliche Beschränkungen in das Gesamtsystem ein. Diese müssen auf Kollision mit anderen Anforderungen überprüft werden.	- Plugin-Systeme (z.B. über das Managment Extensibility Framework) sind potenzielle Sicherheitslücken. - Know-How-Schutz mittels Obfuskation und Reflektion sind nicht kompatibel.
"No Silver Bullet"	Mehrere Lösungen um das selbe "Ziel" zu erreichen erhöhen den Umfang des Projektes, aber nicht zwangsläufig die Komplexität. Wichtig ist es genau zu definieren, in welchem Szenario welche Lösung zu verwenden ist. In der Softwareentwicklung gibt es immer wieder ähnliche Problemstellungen. Werden bestehende Lösungen auf neue Aufgabenstellungen projiziert werden, führt dies im Allgemeinen zu einer suboptimalen Lösung.	Um Abhängigkeiten zwischen (Domänen-)Objekten aufzulösen empfehlen sich mehrere Lösungen - Entkoppelung mittels Events - Entkoppelung mittels Weak-Events - Entkopellung mittels Vermittler-Pattern - Entkopllung mittels Überwacher-Pattern in verbindung mit Vermittler-Pattern. In einem komplexen Softwaresystem ist es wahrscheinlich, dass alle Varianten benötigt werden.
Nutzen sie die Vorteile der eingesetzten Technologien.	Moderne Technologien haben in der Regel einen Mehrwert gegenüber bestehenden Technologien. Die "Gefahr" der Verwendung einer neuen Technologie sollte nur eingegangen werden, wenn der Mehrwert auch verwendet wird.	Aktuelle Oberflächentechnologien besitzen als Hauptvorteil die Testbarkeit (auf Kosten der Laufzeit). Wenn die Testbarkeit nicht genutzt oder nicht benötigt wird, ist der Einsatz der Technologie weniger gerechtfertigt.

Prinzip	Regeln	Beispiel
Seien Sie sich der Nachteile der eingesetzten Technologien bewusst.	Schon während des Architektur-Entwurfs sollten die Schwächen der eingesetzten Technologie beleuchtet werden.	Siehe Abbildung 6.1
Beziehen sie die Kompetenzen des Entwicklungsteams in Architekturentscheidungen mit ein.	Die beste Architektur wird nicht funktionieren, wenn die Entwicklung nicht dahinter steht oder sie nicht versteht.	WPF oder Silverlight ermöglichen das saubere Entwickeln von Oberflächen. Der Einarbeitungsaufwand ist aber enorm. In einem Projekt mit 6 Monaten Laufzeit könnte WPF oder Silverlight nur eingesetzt werden, wenn die Entwicklung die Technologien beherrscht.
Schaffen Sie eine Basis, auf der spätere Entscheidungen getroffen werden können.	Kein Großprojekt wird es schaffen ohne Laufzeit- oder Speicherproblemen zum Kunden geliefert zu werden. Ein einheitliches Tracing-Konzept ermöglicht es, Vorgänge im System zu protokollieren und zu bewerten.	Beginnend von einer Benutzeraktion sollte Tracing folgende Fragen beantworten können: - Welche (Domänen-)Objekte werden geladen - Welche Daten werden an den Objekten abgefragt - Welche Methoden werden an welchem Objekt ausgeführt - Wie lange dauert es pro Objekt - Wie viel Speicher ist nach der Aktion mehr belegt als vorher Dies ermöglicht es, Problempunkte zu identifizieren und eine Optimierung des 90%-Falles vorzunehmen.

Prinzip	Regeln	Beispiel
Definieren Sie realistische Benchmark-Szenarien und Zielwerte.	Ohne Zielvorgaben (maximaler Speicherverbrauch, durchschnittlicher Speicherverbrauch, maximale Laufzeit pro Aktion, maximale Anzahl Domänenobjekte gleichzeitig geladen, usw.) und Überprüfung dieser Vorgaben werden Probleme zu spät erkannt.	Ein Test-Rahmen mittels automatisiertem Testing (NUnit, MsTest oder andere) ermöglicht die Kontrolle der Benchmark-Vorgaben.
Planen Sie von Begin Abstrahierungsschichten ein.	Zugriffe auf Betriebssystem-Ressourcen (Dateisystem, Bitmaps), "Teure Ressourcen" (Datenbank-, TCP-Verbindungen) oder auf Third-Party-Komponenten sollten nur über eigene Wrapper erfolgen. Nur durch die Abstrahierung von der direkten Implementierung ist eine spätere Anpassung (Optimierung) bzw. der Austausch von Komponenten möglich.	Vielleicht muss später ein virtuelles Dateisystem implementiert werden, die Komponente zum Rendern von Druck-Daten ausgetauscht werden oder Caching von nativen Ressourcen eingeführt werden.
Betreiben Sie aktive Wiederverwendung.	Bestehende Open Source oder Third Party Komponenten sind meist die bessere Lösung. Für eine Selbstimplementierung sollte man sich nur entscheiden, wenn die Anforderungen sonst nicht realisierbar sind.	Für Oberflächenelemente, Tracing-Komponente, OR-Mapper und Datenbanken gibt es erprobte Lösungen fertig zu kaufen. Eine Selbstimplementierung lenkt im Allgemeinen von der eigentlichen Problemstellung der Software ab und schafft unnötige Risiken. Abstrahierung darf nicht vergessen werden!

Prinzip	Regeln	Beispiel
Deklarative Ansätze entkoppeln Logik und Realisierung	Im Einsatz muss aber beides zusammen funktionieren. Deklarative Ansätze im Kontext der Testbarkeit oder der Fehleranalyse müssen untersucht werden, besonders im Hinblick auf (Post-Mortem-) Fehleranalyse beim Kunden. Betrachten Sie die Vorteile des deklarativen Ansatzes, besonders in Zusammenwirkung mit Dependency Injektion.	Wenn Compiler-Unterstützung durch Binding-By-Name ersetzt wird, sollte es ein Tooling geben, welches den Nachteil der Fehleranfälligkeit behebt.
Halten Sie ihre Randbedingungen, Festlegungen und ähnliches in verbindlichen Regeln fest.	Alles was nicht verboten ist, ist erlaubt. Dieser einfache Grundsatz gilt, wenn bei der Entwicklung der Architektur von bestimmten Rahmenbedingungen ausgegangen wird. Diese sollten in Form von Regeln definiert und der Quelltext regelmäßig untersucht werden.	Werkzeuge wie FxCop ermöglichen Überprüfungen des Quelltextes und Definition eigener Regeln.
Arbeiten Sie Iterative und validieren Sie regelmäßig die Architektur anhand der Anforderungen.	So wie "Rom nicht an einem Tag erbaut wurde", ist es auch nicht möglich eine gute Softwarearchitektur an einem Tag zu entwerfen.	Klassifizieren Sie die Anforderungen nach Wichtigkeit und integrieren erst die wichtigsten Anforderungen in die Architektur. Beschränkungen, welche sich daraus ergeben, sind somit verbindlich für die Architektur. Arbeiten Sie anschließend immer wieder einige weitere Anforderungen ein. Nach jeder iteration sollte die Architektur auf Schwächen bzgl. der eingesetzten Technologien überprüft werden.

Tabelle 6.1: Leitfaden für den Softwarearchitektur-Entwurf

6.2 Erkenntnisse

Die im Rahmen der Aufgabenstellung aufgestellten Thesen wurden in den letzten Kapiteln mehr oder weniger direkt erörtert. Zusammenfassend kann man sagen:

These 1 stimmt. Ohne Beachtung der eingesetzten Technologie wird die Architektur (gerade für Großprojekte) nicht tragen. Das .NET Framework bietet mit Reflektion einem Softwarearchitekten viele Freiheitsgrade beim Entwurf der Architektur. Reflektion ermöglicht einen deskriptiven Ansatz bei der Realisierung der Software. Zum Beispiel lassen sich Funktionalitäten, welche erst zur Laufzeit an die Objekte gebunden werden, realisieren (siehe Kapitel 2.2.2). Auch verwenden alle aktuellen Oberflächentechnologien des .NET Frameworks Reflektion (vgl. Kapitel 3.4.2, 3.4.3, 3.4.4). Diese Variabilität geht auf Kosten der Laufzeit und der Möglichkeit, Fehler einfach zu erkennen. Kapitel 5.1.4 beschreibt Probleme und Lösungsvorschläge beim Einsatz von deskriptiven Ansätzen.
Reflektion hat auch andere Auswirkungen auf die Architektur. Der Einsatz von Aspektorientierung (Kapitel 2.1.3) oder Know-how Schutz mittels Obfuskation sind nicht ohne Weiteres möglich (siehe Kapitel 5.2.2).
Ein weiteres Beispiel dafür, dass die eingesetzte Technologie einen Einfluss auf die Architektur hat, liefert Microsoft, indem sie eigene Architekturmuster wie Model-View-ViewModel (siehe Kapitel 2.2.3) definieren, um eine optimale Architektur in Verbindung mit Ihren aktuellen Oberflächentechnologien zu gewährleisten.
Ein interessantes und gefährliches Beispiel für den Einfluss einer Technologie auf die Architektur beschreibt das Kapitel 5.2.3. Dadurch, dass es im .NET Framework in Verbindung mit dem Visual Studio einfach herauszufinden ist, was für ein Objekt hinter einem bestimmten Interface steht und welche weiteren Interfaces das Objekt noch implementiert, kann vom Softwareentwickler in ein anderes Interface gecastet werden. In diesem Fall wäre der Dialog mit dem Softwarearchitekten zu suchen, weil eventuell die Schnittstellen Definitionen nicht ausreichend sind.
Das Kapitel 2 beschreibt Grundlagen, die einem Softwarearchitekten im .NET-Umfeld bekannt seien sollten.

These 2 stimmt. Viele Entscheidungen sind für "normale" Anwendungen passend. Für die Realisierung von komplexen Softwaresystemen hingegen könnten einige zusätzliche Probleme auftreten. Objekte könnten unter anderem so viel Speicher benötigen oder in so hoher Anzahl gebraucht werden, dass das Laden sehr lange dauert (siehe Kapitel 5.1.1). Durch die Realisierung von Funktionalitäten kommt es zu Abhängigkeiten von Objekten, es ist eine zentrale Architekturentscheidung, wie diese Abhängigkeiten aufzulösen sind. Im schlimmsten Fall sind die Abhängigkeiten über Events aufgelöst, ohne dass es die Möglichkeit gibt, dass Objekte sich von Events wieder abmelden. Dies führt dazu, dass das ganze Objekt-Geflecht (die Objekt-Wolke) im Speicher gehalten wird. Kapitel 5.1.2 beschreibt verschiedene Varianten, um die Abhängigkeiten zwischen Objekten aufzulösen. Das im Kapitel 5.1.3 skizzierte Problem, dass Objekte eventuell Informationen von

anderen Objekten zum Arbeiten brauchen und wie sie an diese kommen, ist in normalen Anwendungen meist zu vernachlässigen. Komplexe Softwaresysteme können nicht alle Objekte gleichzeitig im Speicher halten. Deswegen führt häufiges Navigieren zu sehr schlechtem Laufzeitverhalten. In komplexen Softwaresystemen werden Informationen (z.B. Typeninformationen) in Metadateien abgelegt. Um die Fehleranfälligkeit zu reduzieren und weniger Speicher zu verbrauchen, sind die Informationen meist über mehrere Dateien baummäßig aufgebaut. Die Laufzeit, gerade beim Laden von vielen kleinen Dateien, ist nicht zu vernachlässigen (siehe Kapitel 2.5.2). Eine Unterstützung zum Entwurf von Architekturen für komplexe Softwaresysteme ist die Frage "Kommt meine Architektur/Implementierung mit 500000 Objekten zurecht?". Das Kapitel 2.5.1 beschreibt diese Unterstützung.

These 3 stimmt zum Teil. Mit automatischer Speicherverwaltung ist die Realisierung von großen Softwareprojekten möglich. Doch im Vergleich zum Arbeiten ohne automatische Speicherverwaltung wird die Softwareentwicklung nicht einfacher. Das Kapitel 2.4.1 beinhaltet Wissen, welches benötigt wird, um mit der automatischen Speicherverwaltung des .NET effektiv zu arbeiten. Bei einem Framework ohne Speicherverwaltung besteht die Aufgabe der Entwicklung herauszufinden, welche Objekte Speicherbereiche nicht freigeben. Im Gegensatz dazu müssen beim Einsatz von automatischer Speicherverwaltung die Objekte gefunden werden, die Speicherbereiche stark referenzieren.

Ein Problem, das es so nur im .NET-Umfeld gibt, ist der Large Object Heap (LOH). Auf selbigem werden Objekte gespeichert, die wahrscheinlich lange leben. Dass die Objekte wahrscheinlich lange leben entscheidet das .NET Framework anhand der Größe der Objekte. Bei mehr als 85KB werden die Objekte auf dem LOH gespeichert. Das Problem ist, dass der LOH nicht defragmentiert wird. Dadurch ist es möglich mit sehr wenig allokiertem Speicher keinen weiteren Speicherbereich zu finden, der groß genug ist. Dies führt zum Absturz der Software. Das Kapitel A.1.1 zeigt ein Beispiel.

In komplexen Softwaresystemen, die viele Daten verarbeiten, gibt es eine Reihe von Thematiken, die im Zusammenspiel mit automatischer Speicherverwaltung zu Problemen führen:

- Die Standard- Implementierungen der Listen im .NET sind für große Datenmengen nicht geeignet (siehe Kapitel 2.4.2). Der Softwarearchitekt sollte einplanen, eigene Datenstrukturen implementieren zu lassen, die den LOH nicht belasten (siehe Kapitel A.1.3)
- Das .NET Framework arbeitet mit Implicit Invokation. Die dafür genutzten Events referenzieren die angemeldeten Objekte stark, wodurch sie am Leben gehalten werden (siehe Kapitel 2.4.3). Damit dies nicht zum Problem wird, muss der Lebenszyklus von Objekten definiert werden (siehe Kapitel 2.4.4).
- Native Ressourcen belegen Speicher im Prozessraum, welcher der Spei-

cherverwaltung unbekannt ist. Das kann dazu führen, dass beim Aufräumen der Speicherverwaltung der verwaltete Wrapper des .NET Framework noch im Speicher existiert, obwohl der Speicher der nativen Ressource bereits freigegeben wurde. Kapitel 2.4.6 behandelt dieses Thema und beschreibt, wie eigene Wrapper erstellt werden können, die das Problem lösen.

- Zeichenketten sind unveränderlich. Das bedeutet, dass jede Modifikation an einer Zeichenkette dafür sorgt, dass eine Zweite (modifizierte) Zeichenkette im Speicher existiert (siehe Kapitel 2.4.5). Des Weiteren sind Vergleiche von Zeichenketten eine zeitintensive Operation, welche durch String-Interning beschleunigt werden kann. String-Interning hat den Nachteil, dass einmal geladene Zeichenketten nicht mehr aus dem Speicher entfernt werden können. Als Lösung gilt der Einsatz einer eigenen Implementierung des String-Interning. Eine Beispiel- Implementierung befindet sich im Kapitel A.1.4.

These 4 stimmt für das Standard .NET Framework nur zum Teil. Für das Compact .NET Framework oder die WinRT (siehe Kapitel **??**) stimmt diese These jedoch nicht. Viele Betriebssystemfunktionen sind bereits im .NET gekapselt, jedoch nicht alle. In Großprojekten müssen in der Regel mehr Betriebssystemfunktionen verwendet werden als in Small Business Applications. Zum Beispiel existierte bis zum .NET 4.0 keine Möglichkeit verwaltete Memory-Mapped-Files zu verwenden.

Der Zugriff auf native Ressourcen (wie z.B. Betriebssystemfunktionen), wird im Kapitel 2.4.6 beschrieben. Durch die Verwendung von nativen Ressourcen wird eine anpassungsfreie Portierung auf andere nicht Windows- Betriebssysteme verhindert. Das Kapitel 5.2.1 beschreibt Nachteile des Zugriffes auf nativen Ressourcen.

Im Allgemeinen gibt es bei der Realisierung einer Software Optimierungen für das Laufzeit- und Speicherverhalten. Ob diese beim Durchlaufen der Anwendung unter Mono zusätzliche Probleme erzeugen, ist davon abhängig, ob Mono dieselben "Schwächen" aufweist.

6.3 Ausblick

Zukünftige wissenschaftliche Arbeiten könnten sich folgender Punkten annehmen:

- Arbeitet die automatische Speicherverwaltung von Mono anders als der Garbage Collector des .NET Frameworks (besonders in Hinblick auf den Large Object Heap)? Ist Mono vielleicht besser zur Realisierung für komplexe Softwaresysteme geeignet.
- Wie ist es möglich Reflektion in der Anwendung zu verwenden und trotzdem mittels Obfuskation das Know-how zu schützen?
- Ist es möglich mit HTML 5 und CSS, Anwendungen sowohl für den Desktop als auch für das Web zu entwickeln?

Anhang A: Anhang

A.1 Quellcodes

A.1.1 Anwendung entworfen nach MVVM

View

TreeSearch - XAML

```
<UserControl x:Class="DA_MVVM_DEMO.View.TreeSearch"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml
                 /presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/
                 xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-
                 compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/
                 blend/2008"
             xmlns:local="clr-namespace:DA_MVVM_DEMO.ViewModel"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid Name="LayoutRoot" Background="Black">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="25" />
        </Grid.RowDefinitions>
        <TreeView Grid.Row="0"
                  Background="Orange"
                  ItemsSource="{Binding␣FirstLevelElements}">
            <TreeView.ItemContainerStyle>
                <Style TargetType="{x:Type␣TreeViewItem}">
                    <Setter Property="IsExpanded"
                            Value="{Binding␣IsExpanded,␣Mode=
                                TwoWay}" />
                    <Setter Property="IsSelected"
                            Value="{Binding␣IsSelected,␣Mode=
                                TwoWay}" />
                    <Setter Property="FontWeight"
                            Value="Normal" />
                    <Style.Triggers>
```

```
                        <Trigger Property="IsSelected"
                                 Value="True">
                            <Setter Property="FontWeight"
                                    Value="Bold" />
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </TreeView.ItemContainerStyle>
            <TreeView.ItemTemplate>
                <HierarchicalDataTemplate ItemsSource="{Binding␣
                    Elements}">
                    <TextBlock Text="{Binding␣Text}" />
                </HierarchicalDataTemplate>
            </TreeView.ItemTemplate>
        </TreeView>
        <Grid Name="InnerGrid" Grid.Row="1">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="0.8*" />
                <ColumnDefinition Width="0.2*" />
            </Grid.ColumnDefinitions>
            <TextBox Background="DarkBlue"
                     Foreground="White"
                     Grid.Column="0"
                     Text="{Binding␣TextToSearch,
␣␣␣␣␣␣␣␣␣␣␣␣UpdateSourceTrigger=PropertyChanged}"
                     KeyDown="TextBox_KeyDown"/>
            <Button Background="Yellow"
                    Foreground="DarkMagenta"
                    Grid.Column="1"
                    Content="Find"
                    Command="{Binding␣ViewModelSearchingCommand}
                        "/>
        </Grid>
    </Grid>
</UserControl>
```

Listing A.1: MVVM Beispiel: XAML der View

TreeSearch - CodeBehind

```
1 public partial class TreeSearch : UserControl
2 {
3   public TreeSearch()
```

```
4   {
5     DataContext = new TreeSearchViewModel();
6   }
7   private void TextBox_KeyDown(object sender, KeyEventArgs e)
8   {
9     if (e.Key == Key.Enter)
10    {
11      var vm = DataContext as TreeSearchViewModel
12      vm.ViewModelSearchingCommand.Execute(null);
13    }
14  }
15 }
```

Listing A.2: MVVM Beispiel: Codebehind der View

ViewModel

TreeSearchViewModel

```
1  public class TreeSearchViewModel
2  {
3    ////////////////////////////////////////////////////////
4    #region Field(s)
5    private TreeSearchModel m_Model;
6    private TreeSearchModelElement m_ModelRoot;
7    private ViewModelElement m_ViewRoot;
8    private ViewModelSearchCommand m_SearchCommand;
9    private ICollection<ViewModelElement> m_FirstLevelElements;
10   private string m_TextToSearch;
11   #endregion
12   ////////////////////////////////////////////////////////
13   #region Properties
14   public ICollection<ViewModelElement> FirstLevelElements
15   {
16     get { return m_FirstLevelElements; }
17   }
18   public ICommand ViewModelSearchingCommand
19   {
20     get { return m_SearchCommand; }
21   }
22   public string TextToSearch
23   {
24     get { return m_TextToSearch; }
25     set
26     {
```

```
27        if (string.Compare(m_TextToSearch, value) != 0)
28        {
29          m_TextToSearch = value;
30        }
31      }
32    }
33    #endregion
34    ////////////////////////////////////////////////////////
35    #region c'tor
36    public TreeSearchViewModel()
37    {
38      m_Model = new TreeSearchModel();
39      m_ModelRoot = m_Model.GetAllRoots()[0];
40      m_ViewRoot = new ViewModelElement(m_ModelRoot);
41      m_SearchCommand = new ViewModelSearchCommand(this);
42      m_FirstLevelElements = m_ViewRoot.Elements;
43    }
44    #endregion
45    ////////////////////////////////////////////////////////
46    #region Public Implementation
47    public void Search()
48    {
49      var matches = FindMatches(m_TextToSearch, m_ViewRoot);
50      foreach (var match in matches)
51      {
52        match.IsSelected = true;
53        ViewModelElement parent = match.Parent;
54        while (parent != null)
55        {
56          parent.IsExpanded = true;
57          parent = parent.Parent;
58        }
59      }
60    }
61    #endregion
62    ////////////////////////////////////////////////////////
63    #region Private Implementation
64    public IEnumerable<ViewModelElement> FindMatches(string token,
        ViewModelElement element)
65    {
66      if (element.Text.Contains(token))
67        yield return element;
68
69      foreach (ViewModelElement child in element.Elements)
70      {
71        foreach (ViewModelElement match in FindMatches(token, child))
```

```
72        {
73          yield return match;
74        }
75      }
76    }
77    #endregion
78  }
```

Listing A.3: MVVM Beispiel: mögliche Implementierung des ViewModels

ViewModelElement

```
1   public class ViewModelElement : INotifyPropertyChanged
2   {
3     //////////////////////////////////////////////////////
4     #region Field(s)
5     private ViewModelElement m_Parent;
6     private TreeSearchModelElement m_Model;
7     private ICollection<ViewModelElement> m_Elements;
8     private bool m_IsSelected;
9     private bool m_IsExpanded;
10    #endregion
11    //////////////////////////////////////////////////////
12    #region Events & Delegates
13    public event PropertyChangedEventHandler PropertyChanged;
14    #endregion
15    //////////////////////////////////////////////////////
16    #region Properties
17    public ViewModelElement Parent
18    {
19      get { return m_Parent; }
20    }
21    public bool IsSelected
22    {
23      get { return m_IsSelected; }
24      set
25      {
26        if (value != m_IsSelected)
27        {
28          m_IsSelected = value;
29          this.RisePropertyChanged("IsSelected");
30        }
31      }
32    }
33    public bool IsExpanded
```

```
34  {
35    get { return m_IsExpanded; }
36    set
37    {
38      if (value != m_IsExpanded)
39      {
40        m_IsExpanded = value;
41        this.RisePropertyChanged("IsExpanded");
42      }
43    }
44  }
45  public string Text
46  {
47    get
48    {
49      return m_Model.FirstText + " " + m_Model.SecondText;
50    }
51  }
52  public ICollection<ViewModelElement> Elements
53  {
54    get { return m_Elements; }
55  }
56  public bool ThrowOnInvalidConfiguration { get; set; }
57  #endregion
58  //////////////////////////////////////////////////////
59  #region c'tor
60  public ViewModelElement(TreeSearchModelElement person)
61    : this(person, null)
62  {
63  }
64  private ViewModelElement(TreeSearchModelElement person,
65    ViewModelElement parent)
66  {
67    m_Model = person;
68    m_Parent = parent;
69
70    m_Elements = (from child in m_Model.GetAllChilds()
71            select new ViewModelElement(child, this))
72            .ToList<ViewModelElement>();
73  }
74  #endregion
75  //////////////////////////////////////////////////////
76  #region Private Implementation
77  private void RisePropertyChanged(string name)
78  {
79    VerifyPropertyName(name);
```

```
80      if (PropertyChanged != null)
81      {
82        PropertyChanged.Invoke(this,
83          new PropertyChangedEventArgs(name));
84      }
85    }
86    [Conditional("DEBUG")]
87    [DebuggerStepThrough]
88    private void VerifyPropertyName(string propertyName)
89    {
90      if (TypeDescriptor.GetProperties(this)[propertyName] == null)
91      {
92        const string m_ErrorMessage = "Invalid property name: {0}";
93        if (ThrowOnInvalidConfiguration)
94        {
95          throw new Exception(
96            string.Format(m_ErrorMessage, propertyName));
97        }
98        else
99        {
100         Debug.Fail(
101           string.Format(m_ErrorMessage, propertyName));
102       }
103     }
104   }
105   #endregion
106 }
```

Listing A.4: MVVM Beispiel: Element als Teil des ViewModels

ViewModelSearchCommand

```
1  public class ViewModelSearchCommand:ICommand
2  {
3    private TreeSearchViewModel m_ViewModel;
4    public ViewModelSearchCommand(TreeSearchViewModel viewModel)
5    {
6      m_ViewModel = viewModel;
7    }
8    public bool CanExecute(object parameter)
9    {
10     return true;
11   }
12   public event EventHandler CanExecuteChanged
13   {
```

```
14      add { }
15      remove { }
16    }
17    public void Execute(object parameter)
18    {
19      m_ViewModel.Search();
20    }
21  }
```

Listing A.5: MVVM Beispiel: mögliche Implementierung eines Commands.

Model

TreeSearchModel

```
1   class TreeSearchModel
2   {
3     private readonly IList<TreeSearchModelElement> RootElements
4       = new List<TreeSearchModelElement>();
5     public TreeSearchModel()
6     {
7       //asking business logic or database
8       var root = new TreeSearchModelElement
9       {
10        FirstText = "Autoren",
11      };
12      root.AddChild(
13        new TreeSearchModelElement
14        {
15          FirstText="Jeffrey",
16          SecondText="Richter",
17        });
18      root.AddChild(
19        new TreeSearchModelElement
20        {
21          FirstText = "Martin",
22          SecondText = "Fowler",
23        });
24      root.AddChild(
25        new TreeSearchModelElement
26        {
27          FirstText = "Craig",
28          SecondText = "Larman",
29        });
30      var holger = new TreeSearchModelElement
```

```
31      {
32        FirstText = "Holger",
33        SecondText = "Schwichtenberg",
34      };
35      holger.AddChild(
36        new TreeSearchModelElement
37        {
38          FirstText = "Joachim",
39          SecondText = "Fuchs",
40        });
41      root.AddChild(holger);
42      var erich = new TreeSearchModelElement
43      {
44        FirstText = "Erich",
45        SecondText = "Gamma",
46      };
47      erich.AddChild(
48        new TreeSearchModelElement
49        {
50          FirstText = "Richard",
51          SecondText = "Helm",
52        });
53      erich.AddChild(
54        new TreeSearchModelElement
55        {
56          FirstText = "Ralph",
57          SecondText = "Johnson",
58        });
59      erich.AddChild(
60        new TreeSearchModelElement
61        {
62          FirstText = "John",
63          SecondText = "Vlissides",
64        });
65      root.AddChild(erich);
66      RootElements.Add(root);
67    }
68    public IList<TreeSearchModelElement> GetAllRoots()
69    {
70      return new ReadOnlyCollection<TreeSearchModelElement>(RootElements
            );
71    }
72  }
```

Listing A.6: MVVM Beispiel: Simulation eines Models

TreeSearchModelElement

```
public class TreeSearchModelElement
{
  private readonly IList<TreeSearchModelElement> Children =
    new List<TreeSearchModelElement>();
  public string FirstText { get; set; }
  public string SecondText { get; set; }
  public void AddChild(TreeSearchModelElement child)
  {
    Children.Add(child);
  }
  public void RemoveChild(TreeSearchModelElement child)
  {
    if (Children.Contains(child))
    {
      Children.Remove(child);
    }
  }
  public IList<TreeSearchModelElement> GetAllChilds()
  {
    return new ReadOnlyCollection<TreeSearchModelElement>(Children);
  }
}
```

Listing A.7: MVVM Beispiel: Element eines Models

A.1.2 155MB für Out Of Memory

Als Beispiel für ein schnelle OutOfMemoryException dient folgender Quelltext. Er imitiert ein Programm welches zyklisch riesige Daten (über 10MB) einlesen muss. Diese Dateien werden analysiert und die Relevanten Informationen in einem Zeilobjekt gespeichert. Auf Grund der Menge der beinhaltenden Daten sind die Resultobjekte über 85KB gross. In der Zeit wo das Programm eine Datei analysiert wachsen die folgenden Dateien weiter an. Obwohl das Programm den Speicher für die geladenen Datein wieder freigibt, kommt es zwangsweise zu einer Ausnahme. Das geschieht weil nach einer gewissen Zeit der gesammte Heap fragmentiert ist und kein genügend grosser, freier und zusammenhängender Speicherblock gefunden werden kann. Das Beispielprogramm stürzt auf einen 32Bit Rechner bei 165MB allokierten Speicher ab. In sehr ungünstigen Szenarien ist es auch möglich, dass mit weniger als 10MB allkoierten Speicher kein weiterer Speicherblock gefunden werden kann.

```
public class Class1
{
```

```
3     private const int c_SizeOfResult = 85*1024;
4     private const long c_DeltaBigData = 174592;        //170,5KB
5     private const long c_BigDataStartValue = 1*1024*1024; //1MB
6
7     private static void Main(string[] args)
8     {
9         IList<byte[]> m_ResultData = new List<byte[]>();
10        long size = c_BigDataStartValue;
11        long startingMemory = 0;
12        try
13        {
14            startingMemory = GC.GetTotalMemory(true);
15            while (true)
16            {
17                Console.WriteLine("allocating big byte[] size: {0}MB",
18                    size / (1024*1024));
19                //allocated on LoH
20                byte[] bigData = new byte[size];
21                Debug.Assert(GC.GetGeneration(bigData) == 2,
22                    "large object isn't on LoH, why?");
23                byte[] result = new byte[c_SizeOfResult];
24                Debug.Assert(GC.GetGeneration(result) == 2,
25                    "result object isn't on LoH, why?");
26                //strong reference to "small" large object
27                m_ResultData.Add(result);
28                //avoid that array will be eleminated from compiler
29                Foo(bigData);
30                //next object is bigger
31                size += c_DeltaBigData;
32                Console.WriteLine("referenced memory at LoH {0}Bytes",
33                    GC.GetTotalMemory(true) - startingMemory);
34            }
35        }
36        //OutOfMemory Exception will breaking the while loop
37        catch (Exception e)
38        {
39            Console.WriteLine("got exception! {0}", e.Message);
40        }
41        finally
42        {
43            Console.WriteLine("got {0} elements, memory: {1}MB",
44                m_ResultData.Count,
45                GC.GetTotalMemory(true) / (1024 * 1024));
46        }
47    }
48
```

```
49      private static void Foo(byte[] input)
50      {
51          for (int i = 0; i < input.Length; ++i)
52          {
53              input[i] = (byte)0xFF;
54          }
55      }
56  }
```

Listing A.8: Ende des Speichers nach 155 MB.

A.1.3 Einfache LOH freie Collection

```
1   public class DoubleLinkedList<T> : IList<T>
2   {
3     ////////////////////////////////////////////////////////////
4     #region nested classes
5     private class Node<T> : IEquatable<Node<T>>
6     {
7       public Node<T> Prev { get; set; }
8       public Node<T> Next { get; set; }
9       public T Value { get; set; }
10      ////////////////////////////////////////////////////////////
11      #region I... Implementation
12      #region IEquatable<Node> Members
13      public bool Equals(Node<T> other)
14      {
15        bool result = true;
16
17        result &= other != null;
18        if (result)
19        {
20          result &= Value.Equals(other.Value);
21          result &= ReferenceEquals(Next, other.Next);
22          result &= ReferenceEquals(Prev, other.Prev);
23        }
24        return result;
25      }
26      #endregion
27      #endregion
28      ////////////////////////////////////////////////////////////
29      #region Public Implementation
30      public static bool operator ==(Node<T> lhs, Node<T> rhs)
31      {
32        if (ReferenceEquals(lhs, rhs))
33        {
```

```
34        return true;
35      }
36      return lhs.Equals(rhs);
37    }
38
39    public static bool operator !=(Node<T> lhs, Node<T> rhs)
40    {
41      return !(lhs == rhs);
42    }
43    #endregion
44  }
45  #endregion
46  ////////////////////////////////////////////////////////////////
47  #region fields
48  private long m_Count;
49  private Node<T> m_First;
50  private Node<T> m_Last;
51  #endregion
52  ////////////////////////////////////////////////////////////////
53  #region c'tor(s)
54  public DoubleLinkedList()
55  {
56    m_Count = 0;
57  }
58  public DoubleLinkedList(IEnumerable<T> input)
59    : this()
60  {
61    foreach (T val in input)
62    {
63      Add(val);
64    }
65  }
66  #endregion
67  ////////////////////////////////////////////////////////////////
68  #region I... Implemetation
69
70  #region IList<T> Members
71
72  public int IndexOf(T item)
73  {
74    return unchecked((int)InternalIndexOf(item));
75  }
76  public void Insert(int index, T item)
77  {
78    if (index < 0 || index >= m_Count)
79      throw new ArgumentException("index is out of bound");
```

```
80
81      Node<T> oldNode = Search(index);
82      var newNode = new Node<T>
83      {
84        Next = oldNode,
85        Prev = oldNode.Prev,
86        Value = item,
87      };
88
89      if (oldNode == m_First) m_First = newNode;
90
91      if (oldNode.Prev != null) oldNode.Prev.Next = newNode;
92      oldNode.Prev = newNode;
93
94      m_Count++;
95    }
96    public T this[int index]
97    {
98      get
99      {
100       Node<T> node = Search(index);
101       if (node == null)
102         throw new InvalidOperationException
103         ("there is no entry at this index!");
104
105       return node.Value;
106     }
107     set
108     {
109       Node<T> node = Search(index);
110       if (node != null)
111       {
112         node.Value = value;
113       }
114       else
115       {
116         throw new InvalidOperationException
117         ("there is no entry to replace value at this index!");
118       }
119     }
120   }
121   #endregion
122
123   #region ICollection<T> Members
124   public void Add(T item)
125   {
```

```
126     var node = new Node<T>
127     {
128       Value = item,
129     };
130     InternalAdd(node);
131   }
132   public bool Contains(T item)
133   {
134     return InternalEquals(item);
135   }
136   public void CopyTo(T[] array, int arrayIndex)
137   {
138     if ((array.Length - arrayIndex) >= m_Count)
139     {
140       Node<T> node = m_First;
141       while (node != null)
142       {
143         array[arrayIndex++] = node.Value;
144         node = node.Next;
145       }
146     }
147     else
148       throw new ArgumentException
149       ("array is to small to keep all items of list");
150   }
151   public bool Remove(T item)
152   {
153     Tuple<Node<T>, bool> tuple = Search(item);
154     if (tuple.Item2)
155     {
156       InternalRemove(tuple.Item1);
157     }
158     return tuple.Item2;
159   }
160   public void RemoveAt(int index)
161   {
162     if (index < 0 || index >= m_Count)
163       throw new InvalidOperationException
164       ("there is no entry to remove at this index!");
165
166     Node<T> node = Search(index);
167     InternalRemove(node);
168   }
169   public int Count
170   {
171     get { return unchecked((int)m_Count); }
```

```
172   }
173   public bool IsReadOnly
174   {
175     get { return false; }
176   }
177   public void Clear()
178   {
179     m_First = null;
180     m_Last = null;
181     m_Count = 0;
182   }
183   #endregion
184
185   #region IEnumerable Members
186   IEnumerator<T> IEnumerable<T>.GetEnumerator()
187   {
188     Node<T> node = m_First;
189     while (node != null)
190     {
191       yield return node.Value;
192       node = node.Next;
193     }
194   }
195   IEnumerator IEnumerable.GetEnumerator()
196   {
197     return (this as IEnumerable<T>).GetEnumerator();
198   }
199   #endregion
200
201   #endregion
202   ////////////////////////////////////////////////////////////
203   #region Private Implemetation
204   private Tuple<Node<T>, bool> Search(T item)
205   {
206     if (item == null) throw new ArgumentNullException("item");
207     Node<T> node = m_First;
208     bool found = false;
209     while (node != null && !found)
210     {
211       if (node.Value.GetHashCode() == item.GetHashCode())
212         found = true;
213       else
214         node = node.Next;
215     }
216     return new Tuple<Node<T>, bool> (node, found);
217   }
```

```
218   private Node<T> Search(int index)
219   {
220     Node<T> result;
221     int backwardIndex = unchecked((int)(m_Count - index));
222     if (index >= backwardIndex)
223     {
224       result = BackSearch(m_Last, backwardIndex);
225     }
226     else
227     {
228       result = Search(m_First, index);
229     }
230     return result;
231   }
232   private Node<T> Search(Node<T> n, int index)
233   {
234     Node<T> lastNode = null;
235     if (n != null)
236     {
237       lastNode = n;
238       for (int i = 0; i < index; i++)
239       {
240         lastNode = lastNode.Next;
241       }
242     }
243     return lastNode;
244   }
245   private Node<T> BackSearch(Node<T> n, int index)
246   {
247     Node<T> lastNode = null;
248     if (n != null)
249     {
250       lastNode = n;
251       for (int i = 1; i < index; i++)
252       {
253         lastNode = lastNode.Prev;
254       }
255     }
256     return lastNode;
257   }
258   private void InternalRemove(Node<T> n)
259   {
260     Node<T> prev = n.Prev;
261     Node<T> next = n.Next;
262
263     if (prev != null) prev.Next = next;
```

```
264     if (next != null) next.Prev = prev;
265
266     if (m_First != null && m_First == n) m_First = next;
267     if (m_Last != null && m_Last == n) m_Last = prev;
268
269     --m_Count;
270   }
271   private long InternalAdd(Node<T> n)
272   {
273     if (n == null) throw new ArgumentNullException("n");
274     if (m_First == null)
275     {
276       m_First = n;
277     }
278     else
279     {
280       if (m_Last != null)
281       {
282         m_Last.Next = n;
283         n.Prev = m_Last;
284       }
285       m_Last = n;
286       if (m_First.Next == null) m_First.Next = m_Last;
287       if (m_Last.Prev == null) m_Last.Prev = m_First;
288     }
289     return ++m_Count;
290   }
291   private bool InternalEquals(T target)
292   {
293     bool result = false;
294     if (m_First != null)
295     {
296       Node<T> current = m_First;
297       do
298       {
299         result = current.Value.GetHashCode() == target.GetHashCode();
300         current = current.Next;
301       } while (current != null && !result);
302     }
303     return result;
304   }
305   private long InternalIndexOf(T target)
306   {
307     long position = -1;
308     bool found = false;
309
```

```
310     if (m_First != null)
311     {
312       Node<T> current = m_First;
313       do
314       {
315         ++position;
316         found = current.Value.GetHashCode() == target.GetHashCode();
317         current = current.Next;
318       } while (current != null && !found);
319     }
320     return found ? position : -1;
321   }
322   #endregion
323 }
```

Listing A.9: Niemals auf den LOH gespeicherte doppelt verkettete Liste.

A.1.4 String interning ohne dauerhaften Speicherverbrauch

```
1  public class StringHandler
2  {
3    ////////////////////////////////////////////////////////////////
4    #region field(s)
5    private IDictionary<int, WeakReference> m_Dict;
6    private static StringHandler m_Self;
7    #endregion
8    ////////////////////////////////////////////////////////////////
9    #region c'tor(s)
10
11   private StringHandler()
12   {
13     m_Dict = new Dictionary<int, WeakReference>();
14   }
15
16   [MethodImpl(MethodImplOptions.Synchronized)]
17   public static StringHandler Create()
18   {
19     if (m_Self == null)
20     {
21       m_Self = new StringHandler();
22     }
23     return m_Self;
24   }
25   #endregion
26   ////////////////////////////////////////////////////////////////
27   #region Public Implementation
```

```
28    public string Intern(string text)
29    {
30      int key = text.GetHashCode();
31      if (!m_Dict.ContainsKey(key))
32      {
33        m_Dict.Add(key, new WeakReference(text));
34      }
35      return GetString(key, text);
36    }
37
38    public string IsInterned(string text)
39    {
40      int key = text.GetHashCode();
41      bool contained = m_Dict.ContainsKey(key);
42
43      return contained
44        ? GetString(key, text)
45        : null;
46    }
47
48    public void CleanUp()
49    {
50      m_Dict.Clear();
51    }
52    #endregion
53    ////////////////////////////////////////////////////////////////
54    #region Private Implementation
55    private string GetString(int key, string text)
56    {
57      if (!m_Dict[key].IsAlive)
58      {
59        m_Dict[key] = new WeakReference(text);
60      }
61      return (m_Dict[key].Target as string);
62    }
63    #endregion
64  }
```

Listing A.10: String Interning ohne dauerhaften Speicherverbrauch

A.1.5 Graph-Plotter WinForms Usercontrol

```
1   public class Shape: UserControl
2   {
3     ////////////////////////////////////////////////////////////
4     #region field(s)
```

```
5   private IList<Point> m_Points;
6   private Pen m_BlackPen;
7   private Pen m_RedPen;
8   private Font m_Font;
9   private const int m_BorderGap = 40;
10  private const int m_Gap = 20;
11  private const string m_FontFam = "Arial";
12  private const float m_FontSize = 8;
13  private const string m_TextXBorder = "x";
14  private const string m_TextYBorder = "y";
15  private const string m_TextZeroPoint = "0";
16  #endregion
17  //////////////////////////////////////////////////////////
18  #region c'tor
19  public Shape(IList<Point> points)
20  {
21    m_Points = points;
22    m_BlackPen = new Pen(Color.Black);
23    m_RedPen = new Pen(Color.DarkMagenta);
24    m_Font = new Font(m_FontFam, m_FontSize, FontStyle.Regular);
25  }
26  #endregion
27  //////////////////////////////////////////////////////////
28  #region override's
29  protected override void OnPaint(PaintEventArgs e)
30  {
31    base.OnPaint(e);
32
33    var graphic = e.Graphics;
34    //draw area
35    graphic.FillRectangle(Brushes.White, 0, 0, Width, Height);
36    //x
37    graphic.DrawLine(m_BlackPen,
38      m_BorderGap,
39      Height - m_BorderGap,
40      Width - m_BorderGap,
41      Height - m_BorderGap);
42
43    for (int i = m_Gap, x, y; i < Width - 2 * m_BorderGap; i += m_Gap)
44    {
45      x = m_BorderGap + i;
46      y = Height - m_BorderGap;
47      graphic.DrawLine(m_BlackPen,
48        x,
49        y + 5,
50        x,
```

```
51          y - 5);
52
53        graphic.DrawString(i.ToString(),
54          m_Font,
55          Brushes.ForestGreen,
56          x - 5,
57          y + 20);
58      }
59      graphic.DrawString(m_TextXBorder,
60        m_Font,
61        Brushes.Brown,
62        Width - m_BorderGap + 5,
63        Height - m_BorderGap - 5);
64      //y
65      graphic.DrawLine(m_BlackPen,
66        m_BorderGap,
67        m_BorderGap,
68        m_BorderGap,
69        Height - m_BorderGap);
70
71      for (int i = (int)(Height - 2.5 * m_BorderGap) , x, y, step =
          m_Gap;
72        i >= m_Gap;
73        i -= m_Gap)
74      {
75        x = m_BorderGap;
76        y = m_BorderGap + i;
77        graphic.DrawLine(m_BlackPen,
78          x + 5,
79          y,
80          x - 5,
81          y);
82
83        graphic.DrawString(step.ToString(),
84          m_Font,
85          Brushes.ForestGreen,
86          x - 30,
87          y - 5);
88        step += m_Gap;
89      }
90      graphic.DrawString(m_TextYBorder,
91          m_Font,
92          Brushes.Brown,
93          m_BorderGap - 10,
94          m_BorderGap - 10);
95      // zero point
```

```
96      graphic.DrawString(m_TextZeroPoint,
97        m_Font,
98        Brushes.ForestGreen,
99        m_BorderGap - 10,
100       Height - m_BorderGap + 10);
101     //plotting
102     Point lastPoint = new Point();
103     Point basePoint = new Point(m_BorderGap, Height - m_BorderGap);
104
105     foreach (Point p in m_Points)
106     {
107       if (!lastPoint.IsEmpty)
108       {
109         graphic.DrawLine(m_RedPen,
110           basePoint.X + lastPoint.X * 10,
111           basePoint.Y - lastPoint.Y * 10,
112           basePoint.X + p.X * 10,
113           basePoint.Y - p.Y * 10);
114       }
115       else
116       {
117         graphic.DrawLine(m_RedPen,
118           basePoint.X,
119           basePoint.Y,
120           basePoint.X + p.X * 10,
121           basePoint.Y - p.Y * 10);
122       }
123       lastPoint = p;
124     }
125   }
126   protected override void Dispose(bool disposing)
127   {
128     try
129     {
130       if (m_BlackPen != null) m_BlackPen.Dispose();
131       if (m_RedPen != null) m_RedPen.Dispose();
132       if (m_Font != null) m_Font.Dispose();
133     }
134     finally
135     {
136       base.Dispose(disposing);
137     }
138   }
139   #endregion
140 }
```

Listing A.11: Graph-Plotter als WinForms Oberflächenelement

A.1.6 Graph-Plotter WPF Usercontrol

Es gibt verschiendeste Möglichkeiten um eigene Controls in WPF zu designen *Custom-Control*, *UserControl*, *Visual Layer Programming (VAL)* und andere mehr. Das Beispiel wurde mittels VAL umgesetzt, dies ist erst ab dem .NET 4.0 verfügbar, der klare Vorteil ist das die Zeichen-Logik eines bestehenden WinForms Controls fast komplett übernommen werden kann.

XAML der Anwendung

```
<Window x:Class="GraphPlotterTest.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
            presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WPF Graph Plotter" Height="320" Width="320"
        xmlns:customControl="clr-namespace:GraphPlotter;assembly
            =GraphPlotterLibrary">
    <StackPanel HorizontalAlignment="Center">
        <Canvas Name="MyCanvas"
    Height="284"
    Width="300">
            <Canvas.Children>
                <customControl:GraphPlotter
          Name="plotter"
          Height="284"
          Width="300"
          Points="0,0;5,2;7,4;5,7;9,7;18,10;23,15" />
            </Canvas.Children>
        </Canvas>
    </StackPanel>
</Window>
```

Listing A.12: XAML einer WPF-Anwendung (mit Graph-Plotters als WPF-Oberflächenelement).

Programmcode des User Controls

```
1 public class GraphPlotter : FrameworkElement
2 {
3   //////////////////////////////////////////////////////////
4   #region field(s)
```

```
5   private VisualCollection m_Children;
6   private Size m_LastSize;
7   private const int m_BorderGap = 40;
8   private const int m_Gap = 20;
9   private const string m_FontFam = "Arial";
10  private const float m_FontSize = 8;
11  private const string m_TextXBorder = "x";
12  private const string m_TextYBorder = "y";
13  private const string m_TextZeroPoint = "0";
14  private Pen m_BlackPen;
15  private Pen m_RedPen;
16  #endregion
17  //////////////////////////////////////////////////////////
18  #region Properties
19  public IList<Point> DrawPoints
20  {
21    get
22    {
23      return ConvertToPointList(Points);
24    }
25  }
26  public string Points { get; set; }
27  public static DependencyProperty PointProperty = DependencyProperty.
      Register(
28    "Points",                 //Name
29    typeof(string),           //Propertytype
30    typeof(GraphPlotter),     //owner
31    new PropertyMetadata(
32      string.Empty,         //default value
33      new PropertyChangedCallback(OnPointPropertyChanged)));
34  #endregion
35  //////////////////////////////////////////////////////////
36  #region c'tor
37  public GraphPlotter()
38  {
39    m_Children = new VisualCollection(this);
40    m_BlackPen = new Pen(Brushes.Black, 1);
41    m_RedPen = new Pen(Brushes.DarkMagenta, 1);
42  }
43  #endregion
44  //////////////////////////////////////////////////////////
45  #region override's
46  protected override Size ArrangeOverride(Size finalSize)
47  {
48    m_LastSize = finalSize;
49    ReDraw(true);
```

```
50      return base.ArrangeOverride(finalSize);
51    }
52    protected override int VisualChildrenCount
53    {
54      get { return m_Children.Count; }
55    }
56    protected override Visual GetVisualChild(int index)
57    {
58      if (index < 0 || index >= m_Children.Count)
59      {
60        throw new ArgumentOutOfRangeException();
61      }
62      return m_Children[index];
63    }
64    #endregion
65    //////////////////////////////////////////////////////////////
66    #region Private Implementation
67    private static void OnPointPropertyChanged(DependencyObject d,
68          DependencyPropertyChangedEventArgs e)
69    {
70      GraphPlotter plotter = d as GraphPlotter;
71      if (plotter != null)
72      {
73        string newValue = (string)e.NewValue;
74        plotter.Points = newValue;
75        plotter.ReDraw(false);
76      }
77    }
78    private void ReDraw(bool updateAll)
79    {
80      if (updateAll)
81      {
82        m_Children.Clear();
83        m_Children.Add(CreateDrawArea());
84        m_Children.Add(CreateX());
85        m_Children.Add(CreateY());
86        m_Children.Add(CreateZeroPoint());
87        m_Children.Add(CreateGraph());
88      }
89      else if (m_Children.Count > 0)
90      {
91        m_Children[m_Children.Count - 1] = CreateGraph();
92      }
93    }
94    private static IList<Point> ConvertToPointList(string p)
95    {
```

```
96      var list = new List<Point>();
97      foreach (string part in p.Split(new[] { ';' }))
98      {
99        var koord = part.Split(new[] { ',' });
100       double x = double.Parse(koord[0]);
101       double y = double.Parse(koord[1]);
102       list.Add(new Point(x, y));
103     }
104     return list;
105   }
106   private DrawingVisual CreateDrawArea()
107   {
108     var drawingVisual = new DrawingVisual();
109     DrawingContext drawingContext = drawingVisual.RenderOpen();
110     Rect rect = new Rect(
111       new Point(0, 0),
112       new Size(m_LastSize.Width,
113         m_LastSize.Height));
114     drawingContext.DrawRectangle(
115       Brushes.White,
116       (Pen)null, rect);
117
118     drawingContext.Close();
119     return drawingVisual;
120   }
121   private DrawingVisual CreateX()
122   {
123     var drawingVisual = new DrawingVisual();
124     DrawingContext drawingContext = drawingVisual.RenderOpen();
125     drawingContext.DrawLine(
126       m_BlackPen,
127       new Point(m_BorderGap, m_LastSize.Height - m_BorderGap),
128       new Point(m_LastSize.Width - m_BorderGap,
129         m_LastSize.Height - m_BorderGap));
130
131     for (double i = m_Gap, x, y;
132       i < m_LastSize.Width - 2 * m_BorderGap;
133       i += m_Gap)
134     {
135       x = m_BorderGap + i;
136       y = m_LastSize.Height - m_BorderGap;
137       drawingContext.DrawLine(m_BlackPen,
138         new Point(x, y + 5),
139         new Point(x, y - 5));
140
141       drawingContext.DrawText(
```

```
142         GetTextFormatted(i.ToString(), Brushes.ForestGreen),
143       new Point(x - 5, y + 20));
144     }
145
146     drawingContext.DrawText(
147       GetTextFormatted(m_TextXBorder, Brushes.Brown),
148       new Point(m_LastSize.Width - m_BorderGap + 5,
149         m_LastSize.Height - m_BorderGap - 5));
150
151     drawingContext.Close();
152     return drawingVisual;
153   }
154   private DrawingVisual CreateY()
155   {
156     var drawingVisual = new DrawingVisual();
157     DrawingContext drawingContext = drawingVisual.RenderOpen();
158     drawingContext.DrawLine(
159       m_BlackPen,
160       new Point(m_BorderGap, m_BorderGap),
161       new Point(m_BorderGap, m_LastSize.Height - m_BorderGap));
162
163     for (double i = (Height - 2.5 * m_BorderGap), x, y, step = m_Gap;
164       i >= m_Gap;
165       i -= m_Gap)
166     {
167       x = m_BorderGap;
168       y = m_BorderGap + i;
169       drawingContext.DrawLine(
170         m_BlackPen,
171         new Point(x + 5, y),
172         new Point(x - 5, y));
173
174       drawingContext.DrawText(
175         GetTextFormatted(step.ToString(), Brushes.ForestGreen),
176         new Point(x - 30, y - 5));
177       step += m_Gap;
178     }
179
180     drawingContext.DrawText(
181       GetTextFormatted(m_TextYBorder, Brushes.Brown),
182       new Point(m_BorderGap - 10,
183         m_BorderGap - 10));
184
185     drawingContext.Close();
186     return drawingVisual;
187   }
```

```
188   private DrawingVisual CreateZeroPoint()
189   {
190     var drawingVisual = new DrawingVisual();
191     DrawingContext drawingContext = drawingVisual.RenderOpen();
192
193     drawingContext.DrawText(
194       GetTextFormatted(m_TextZeroPoint, Brushes.ForestGreen),
195       new Point(m_BorderGap - 10,
196         m_LastSize.Height - m_BorderGap + 10));
197
198     drawingContext.Close();
199     return drawingVisual;
200   }
201   private Visual CreateGraph()
202   {
203     var drawingVisual = new DrawingVisual();
204     DrawingContext drawingContext = drawingVisual.RenderOpen();
205     bool initial = true;
206     Point lastPoint = new Point();
207     Point basePoint = new Point(m_BorderGap,
208       m_LastSize.Height - m_BorderGap);
209     foreach (Point p in DrawPoints)
210     {
211       if (!initial)
212       {
213         drawingContext.DrawLine(m_RedPen,
214           new Point(basePoint.X + lastPoint.X * 10,
215           basePoint.Y - lastPoint.Y * 10),
216           new Point(basePoint.X + p.X * 10,
217           basePoint.Y - p.Y * 10));
218
219
220       }
221       else
222       {
223         drawingContext.DrawLine(m_RedPen,
224           new Point(basePoint.X,
225           basePoint.Y),
226           new Point(basePoint.X + p.X * 10,
227           basePoint.Y - p.Y * 10));
228
229         initial = false;
230       }
231       lastPoint = p;
232     }
233     drawingContext.DrawText(
```

```
234         GetTextFormatted(m_TextZeroPoint, Brushes.ForestGreen),
235         new Point(m_BorderGap - 10,
236           m_LastSize.Height - m_BorderGap + 10));
237
238       drawingContext.Close();
239       return drawingVisual;
240     }
241     private FormattedText GetTextFormatted(string text, Brush brush)
242     {
243       return new FormattedText(text,
244         CultureInfo.CurrentCulture,
245         FlowDirection.LeftToRight,
246         new Typeface(
247           new FontFamily(m_FontFam),
248           new FontStyle(),
249           new FontWeight(),
250           new FontStretch()),
251           m_FontSize,
252           brush);
253     }
254     #endregion
255 }
```

Listing A.13: Codebehind eines Graph-Plotters als WPF Oberflächenelement

A.1.7 Graph-Plotter METRO Usercontrol

XAML der Anwendung

```
<UserControl x:Class="GraphPlotter.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
    xmlns:ui="using:UI"
    mc:Ignorable="d"
    d:DesignHeight="320" d:DesignWidth="320">
    <Grid x:Name="LayoutRoot"
    Background="White">
        <ui:GraphPlotterControl
      Height="284"
      Width="300"
```

```
        Points="0,0;5,2;7,4;5,7;9,7;18,10;23,15" />
    </Grid>
</UserControl>
```

Listing A.14: XAML einer METRO-Anwendung (mit Graph-Plotter als METRO Oberflächenelement).

XAML des User Controls

```
<UserControl x:Class="UI.GraphPlotterControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
        presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
        compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400"
            Loaded="UserControl_Loaded">
    <Canvas x:Name="LayoutRoot" Background="AliceBlue">
    </Canvas>
</UserControl>
```

Listing A.15: XAML eines Graph-Plotter als METRO-Oberflächenelement.

Programmcode des User Controls

```
1  public sealed partial class GraphPlotterControl : UserControl
2  {
3    //////////////////////////////////////////////////////////////
4    #region field(s)
5    private Size m_LastSize;
6    private const int m_BorderGap = 40;
7    private const int m_Gap = 20;
8    private SolidColorBrush m_BlackBrush;
9    private SolidColorBrush m_GreenBrush;
10   private SolidColorBrush m_BrownBrush;
11   private SolidColorBrush m_RedBrush;
12   private const string m_TextXBorder = "x";
13   private const string m_TextYBorder = "y";
14   private const string m_TextZeroPoint = "0";
15   #endregion
16   //////////////////////////////////////////////////////////////
17   #region Properties
18   private IList<Point> DrawPoints
19   {
```

```
20      get
21      {
22        return ConvertToPointList(Points);
23      }
24    }
25    public string Points { get; set; }
26    public static DependencyProperty PointProperty = DependencyProperty.
        RegisterAttached(
27      "Points",
28      "Object",    //HACK for Win8 Developer Preview
29      typeof(GraphPlotterControl).FullName,
30      new PropertyMetadata(
31        string.Empty,
32        null));      //not needed at demo szenario
33    #endregion
34    //////////////////////////////////////////////////////////////
35    #region c'tor
36    public GraphPlotterControl()
37    {
38      InitializeComponent();
39      m_BlackBrush = new SolidColorBrush(Colors.Black);
40      m_GreenBrush = new SolidColorBrush(Colors.Green);
41      m_BrownBrush = new SolidColorBrush(Colors.Brown);
42      m_RedBrush = new SolidColorBrush(Colors.Magenta);
43    }
44    #endregion
45    //////////////////////////////////////////////////////////////
46    #region override's
47    protected override Size ArrangeOverride(Size finalSize)
48    {
49      m_LastSize = finalSize;
50      return base.ArrangeOverride(finalSize);
51    }
52    #endregion
53    //////////////////////////////////////////////////////////////
54    #region Private Implementation
55    private void UserControl_Loaded(object sender, RoutedEventArgs e)
56    {
57      ReDraw();
58    }
59    private void ReDraw()
60    {
61      LayoutRoot.Children.Clear();
62
63      CreateX();
64      CreateY();
```

```
65      CreateZeroPoint();
66      CreateGraph();
67    }
68    private static IList<Point> ConvertToPointList(string p)
69    {
70      var list = new List<Point>();
71      if (!string.IsNullOrEmpty(p))
72      {
73        foreach (string part in p.Split(new[] { ';' }))
74        {
75          var koord = part.Split(new[] { ',' });
76          double x = double.Parse(koord[0]);
77          double y = double.Parse(koord[1]);
78          list.Add(new Point(x, y));
79        }
80      }
81      return list;
82    }
83    private void CreateX()
84    {
85      var line = new Line
86      {
87        X1 = m_BorderGap,
88        Y1 = m_LastSize.Height - m_BorderGap,
89        X2 = m_LastSize.Width - m_BorderGap,
90        Y2 = m_LastSize.Height - m_BorderGap,
91        Stroke = m_BlackBrush,
92      };
93      LayoutRoot.Children.Add(line);
94      for (double i = m_Gap, x, y;
95        i < m_LastSize.Width - 2 * m_BorderGap;
96        i += m_Gap)
97      {
98        x = m_BorderGap + i;
99        y = m_LastSize.Height - m_BorderGap;
100       var otherLine = new Line
101       {
102         X1 = x,
103         Y1 = y + 5,
104         X2 = x,
105         Y2 = y - 5,
106         Stroke = m_BlackBrush
107       };
108       LayoutRoot.Children.Add(otherLine);
109       var text = new TextBlock
110       {
```

```
111         Text = i.ToString(),
112         Foreground = m_GreenBrush,
113       };
114       Canvas.SetLeft(text, x - 5);
115       Canvas.SetTop(text, y + 20);
116       LayoutRoot.Children.Add(text);
117     }
118     var otherText = new TextBlock
119     {
120       Text = m_TextXBorder,
121       Foreground = m_BrownBrush,
122     };
123     Canvas.SetLeft(otherText,
124       m_LastSize.Width - m_BorderGap + 5);
125     Canvas.SetTop(otherText,
126       m_LastSize.Height - m_BorderGap - 5);
127     LayoutRoot.Children.Add(otherText);
128   }
129   private void CreateY()
130   {
131     var line = new Line
132     {
133       X1 = m_BorderGap,
134       Y1 = m_BorderGap,
135       X2 = m_BorderGap,
136       Y2 = m_LastSize.Height - m_BorderGap,
137       Stroke = m_BlackBrush,
138     };
139     LayoutRoot.Children.Add(line);
140     for (double i = (Height - 2.5 * m_BorderGap), x, y, step = m_Gap;
141       i >= m_Gap;
142       i -= m_Gap)
143     {
144       x = m_BorderGap;
145       y = m_BorderGap + i;
146       var otherLine = new Line
147       {
148         X1 = x + 5,
149         Y1 = y,
150         X2 = x - 5,
151         Y2 = y,
152         Stroke = m_BlackBrush
153       };
154       LayoutRoot.Children.Add(otherLine);
155       var text = new TextBlock
156       {
```

```
157         Text = step.ToString(),
158         Foreground = m_GreenBrush,
159       };
160       Canvas.SetLeft(text, x - 30);
161       Canvas.SetTop(text, y - 5);
162       LayoutRoot.Children.Add(text);
163
164       step += m_Gap;
165     }
166     var otherText = new TextBlock
167     {
168       Text = m_TextYBorder,
169       Foreground = m_BrownBrush,
170     };
171     Canvas.SetLeft(otherText, m_BorderGap - 10);
172     Canvas.SetTop(otherText, m_BorderGap - 10);
173     LayoutRoot.Children.Add(otherText);
174   }
175   private void CreateZeroPoint()
176   {
177     var text = new TextBlock
178     {
179       Text = m_TextZeroPoint,
180       Foreground = m_GreenBrush,
181     };
182     Canvas.SetLeft(text,
183       m_BorderGap - 10);
184     Canvas.SetTop(text,
185       m_LastSize.Height - m_BorderGap + 10);
186     LayoutRoot.Children.Add(text);
187   }
188   private void CreateGraph()
189   {
190     var figure = new PathFigure();
191     bool initial = true;
192     Point basePoint = new Point(m_BorderGap,
193       m_LastSize.Height - m_BorderGap);
194     foreach (Point p in DrawPoints)
195     {
196       if(!initial)
197       {
198         figure.Segments.Add(
199             new LineSegment
200             {
201               Point = new Point(
202                 basePoint.X + p.X * 10,
```

```
203                 basePoint.Y - p.Y * 10),
204             });
205       }
206       else
207       {
208         figure.StartPoint = new Point(
209          basePoint.X,
210          basePoint.Y);
211         figure.Segments.Add(
212           new LineSegment
213           {
214             Point = new Point(
215               basePoint.X + p.X * 10,
216               basePoint.Y - p.Y * 10),
217           });
218       }
219     }
220     var geo = new PathGeometry();
221     geo.Figures.Add(figure);
222     var path = new Path
223     {
224       Data = geo,
225       Stroke = m_RedBrush,
226     };
227     LayoutRoot.Children.Add(path);
228   }
229   #endregion
230 }
```

Listing A.16: Codebehind eines Graph-Plotters als METRO-Oberflächenelement.

A.1.8 Graph-Plotter Silverlight Usercontrol

Der Kontrollfluß bei Silverlight ist anders als bei METRO. Die Methode ArrangeOverride wird zu am Start der Anwendung nicht gleich mit den konkreten Daten aufgerufen, sondern erst mit einer Größe von 0,0. Nach jedem Zeichnen (so auch nach dem ersten) wird ArrangeOverride mit der richtigen Größe aufgerufen, in diesem Fall muss die Oberfläche neu gezeichnet werden. Das ist ziemlich verwirrend, denn im Designer wird der Code zum Zeichnen ausgeführt und dabei kommt auch beim ersten Aufruf von ArrangeOverride die richtige Größe mit. Das Usercontrol im Designer sah so aus wie Abbildung 3.13 im Kapitel Silverlight zeigt. Abbildung A.1 zeigt, wie es aussieht wenn man den Code der METRO-App ohne Veränderungen in Silverlight (im Browser) laufen lässt.

Im Gegensatz zum Quelltext der METRO-App ist nur ein Modifikation der ArrangeOver-

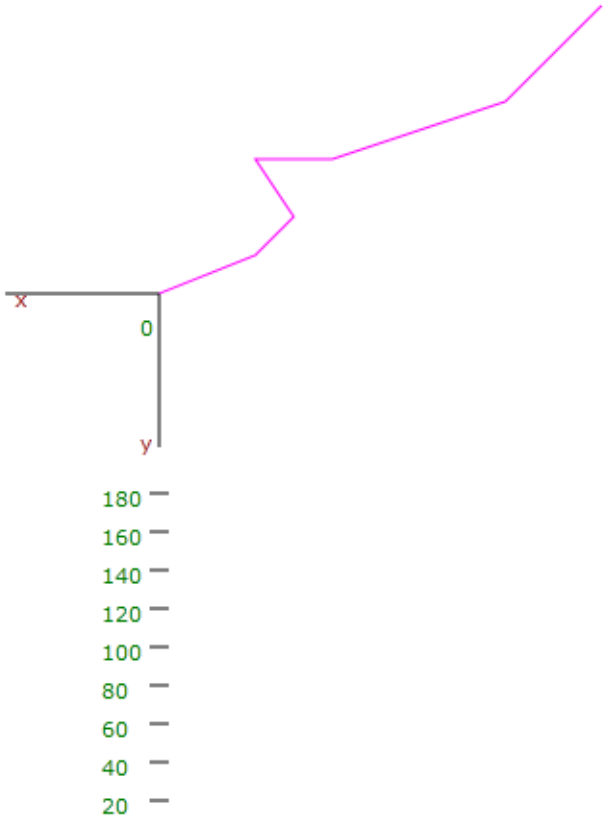

Abbildung A.1: Unterschiede in der Darstellung eines Silverlight UserControl zwischen Browser und Designer

ride Methode notwendig:

```
protected override Size ArrangeOverride(Size finalSize)
{
  if (m_LastSize != finalSize)
  {
    m_LastSize = finalSize;
    ReDraw();
  }
  return base.ArrangeOverride(finalSize);
}
```

Listing A.17: Codebehind eines Graph-Plotters als Silverlight-Oberflächenelement.

A.2 Tutorial: Visual Studio für Entwicklung von verwalteten und nativen Quelltext einrichten

Um im Visual Studio gleichzeitig nativen und verwalteten Quelltext zu entwickeln, müssen die verschiedenen Projekte teil einer Visual Studio Solution (*.sln) sein. Abbildung

A.2 zeigt den Solution Navigator, welcher die Projekte beinhaltet. Abbildung A.3 zeigt

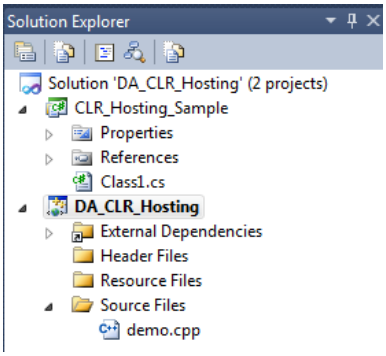

Abbildung A.2: Solution Navigator, zeigt eine Solution mit nativem und verwaltetem Projekten.

wie man den Konfigurationsmanager aufruft. In dem Konfigurationsmanager ist es wich-

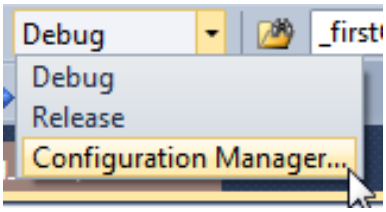

Abbildung A.3: Starten des Konfigurationsmanagers.

tig das alle Projekte in der Solution für dieselbe Zielplattform (x86 & Win32, Itanium, oder x64) generiert werden (siehe Abbildung A.4).

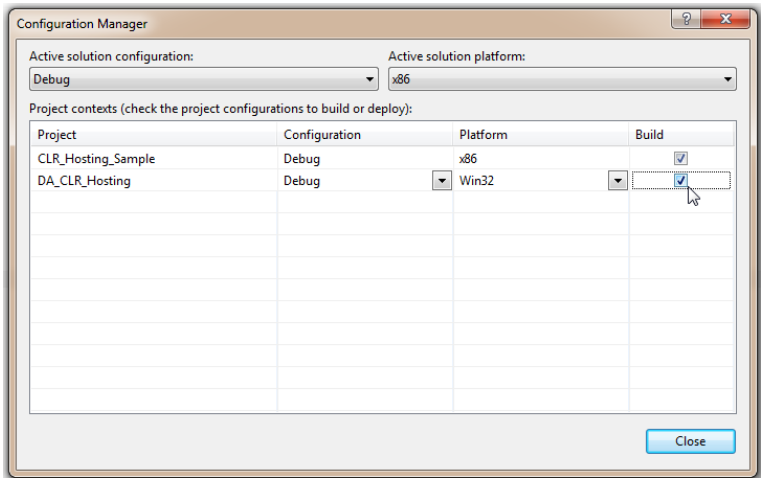

Abbildung A.4: Konfigurationsmanagers, alle Projekte sollten die gleiche Zielplattform haben.

☞ Tipp überprüfen sie ob, nachdem Umstellen der Zielplattform in der Debug-Konfiguration auch noch die DEBUG-Konstanten in den Eigenschaften der Projekte gesetzt sind. Die Eigenschaften eines Projektes erhalten sie, indem sie das Projekt im Solution Navigator selektieren und ALT + Enter drücken. Abbildung A.5 zeigt, wo sie die DEBUG-Konstante für native Projekte finden und Abbildung A.6, wo sich diese bei verwalteten Projekten

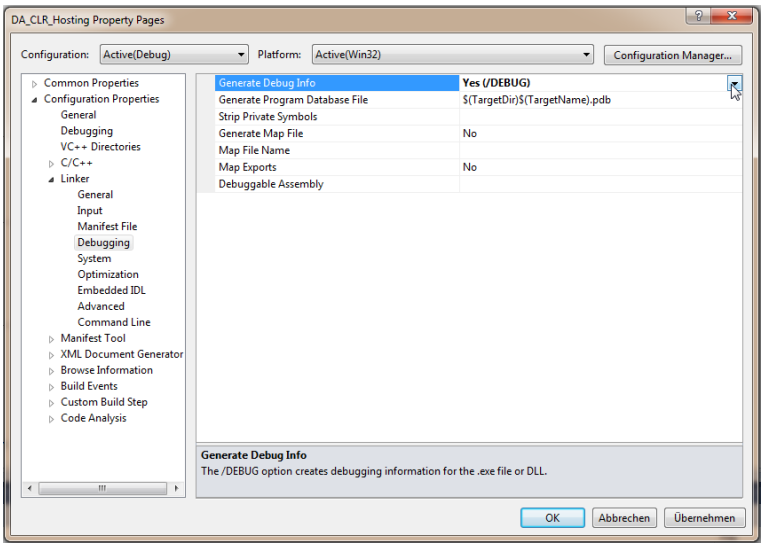

Abbildung A.5: Aktivieren der DEBUG Einstellung für native Projekte.

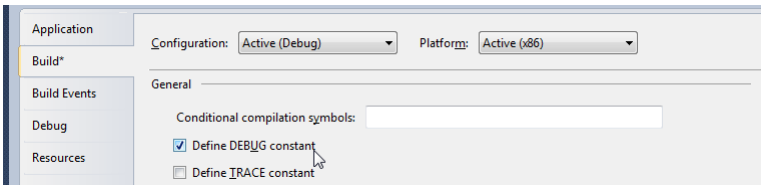

Abbildung A.6: Aktivieren der DEBUG Einstellung für verwaltete Projekte.

befindet.
Jetzt kann man mit dem Visual Studio schon nativ und verwaltet gleichzeitig Entwickeln, ohne weitere Anpassungen an der Entwicklungsumgebung vorzunehmen. Damit auch die Fehlersuche (debugging) funktioniert sind weitere Einstellungen zu tätigen.
Um aus verwaltetem Quelltext in nativen zu debuggen, ist es notwendig die Eigenschaft "Enable unmanged code debugging" zu setzen, dies wird in Abbildung A.7 gezeigt.
Um aus nativem Quelltext in verwalteten zu debuggen, ist es notwendig die Eigenschaft "Debugger Type" auf "Mixed" (nativ und verwaltet) zu setzen, dies wird in Abbildung A.8 gezeigt.
☞ Tipp Mixed-Mode debugging ist nicht möglich auf 64Bit Betriebssystem, siehe `http://msdn.microsoft.com/en-us/library/fz5w87ad%28v=VS.90%29.aspx`.

A.3 Spezialisierung Domänen-Modell

A.3.1 Package - Veranstaltung

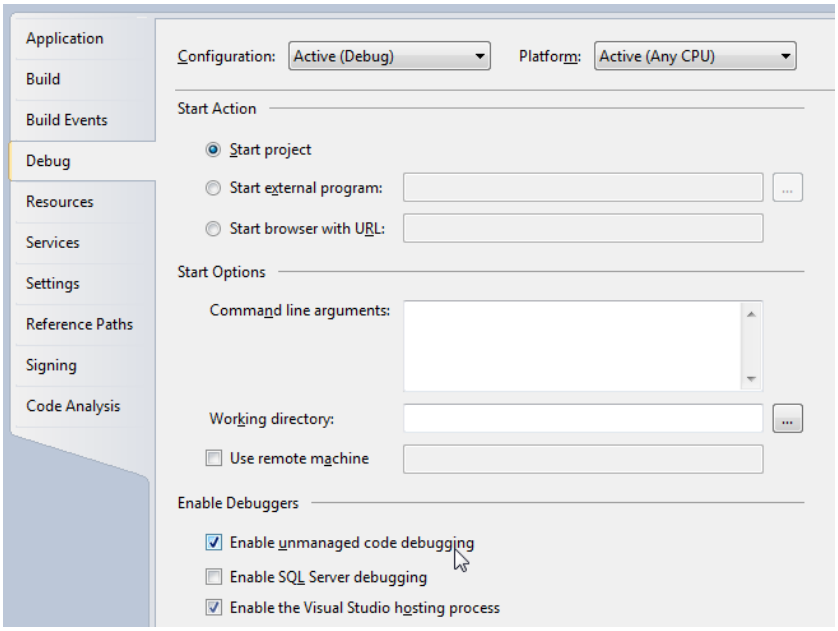

Abbildung A.7: Aktivieren der Einstellung "Enable unmanged code debugging".

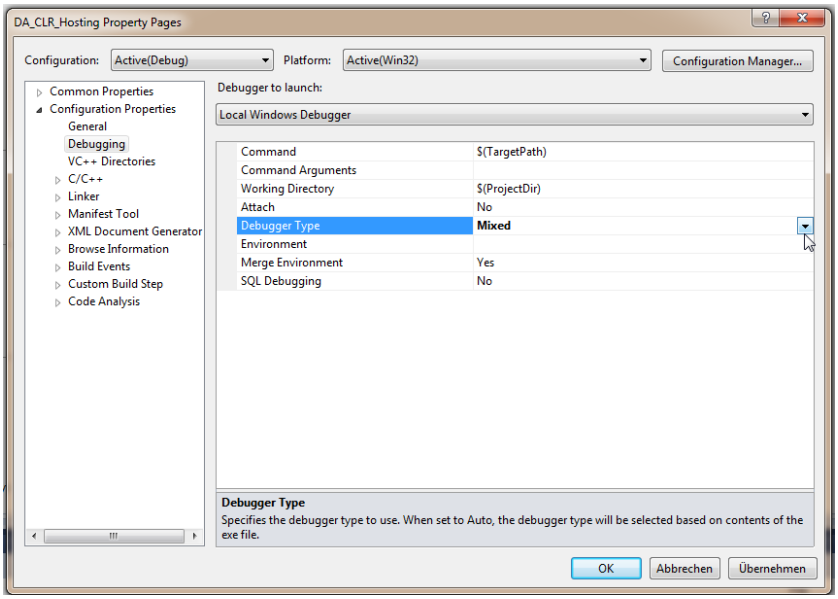

Abbildung A.8: Aktivieren der Einstellung "Enable unmanged code debugging".

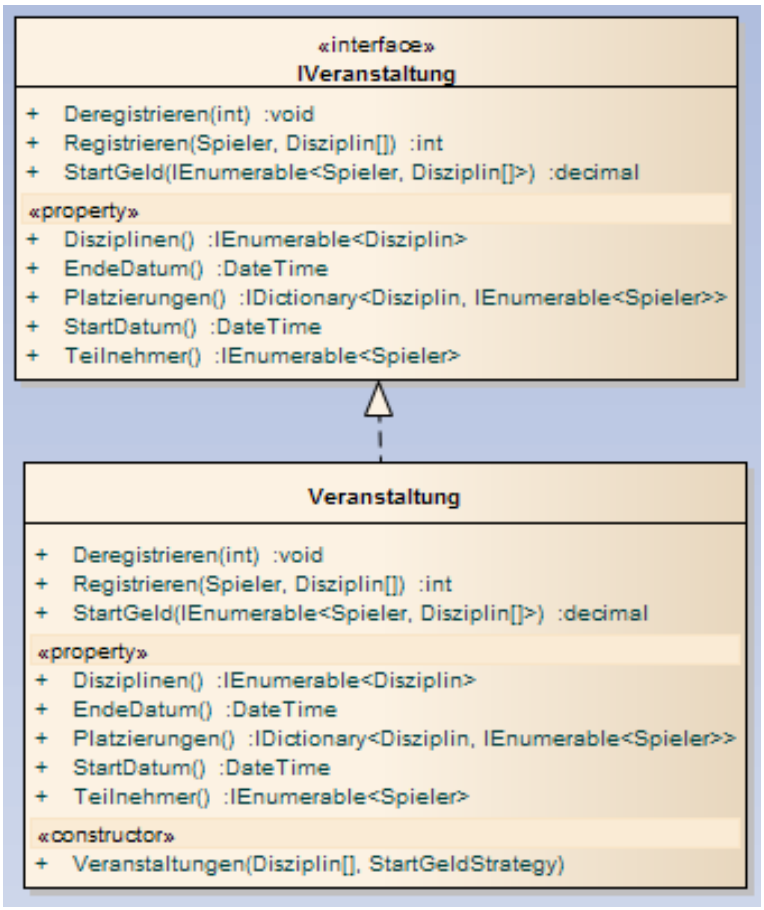

Abbildung A.9: Mögliche Realisierung des Packages "Veranstaltung aus einem Domänen-Modell"

A.3.2 Package - Spieler

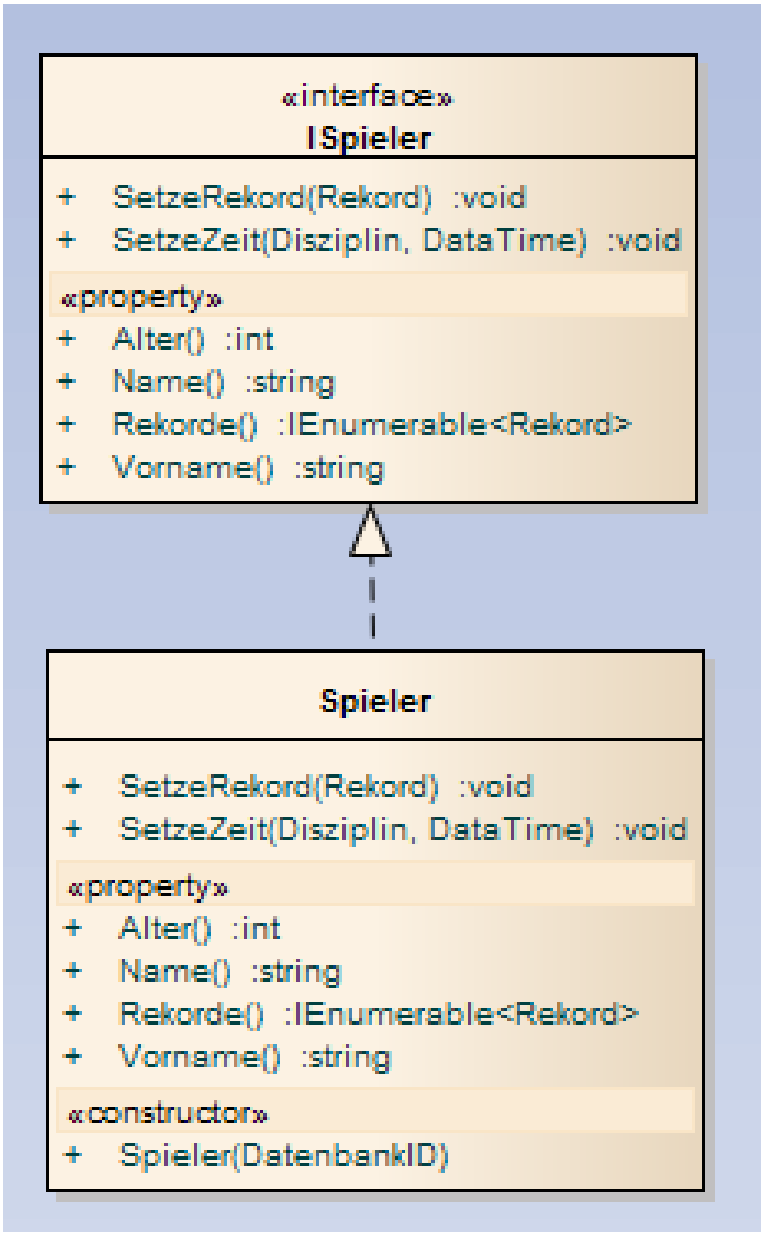

Abbildung A.10: Mögliche Realisierung des Packages "Spieler aus einem Domänen-Modell"

A.3.3 Package - Disziplin

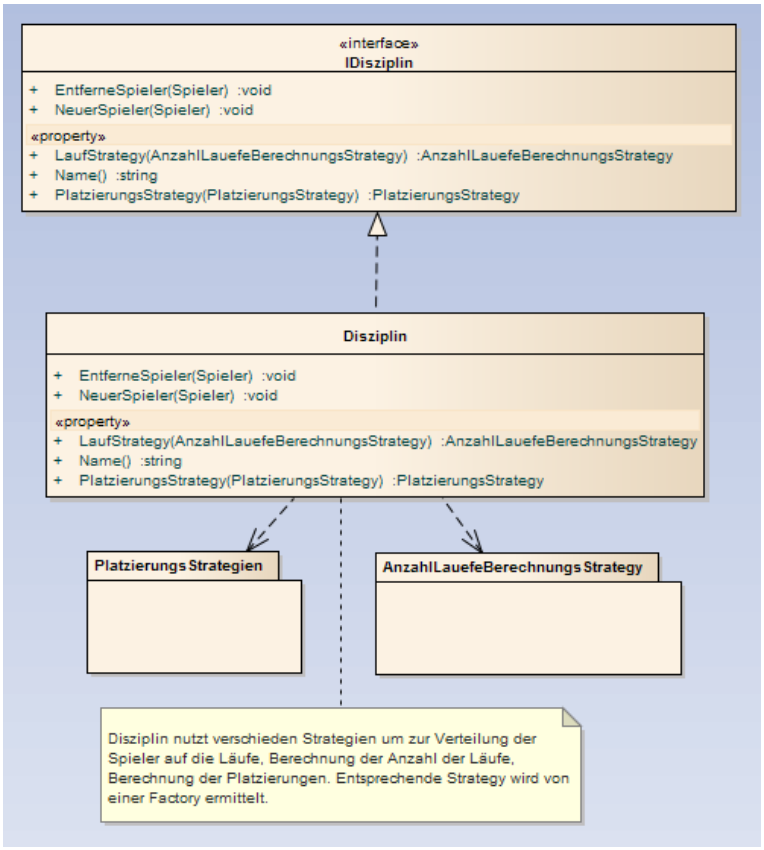

Abbildung A.11: Mögliche Realisierung des Packages "Disziplin aus einem Domänen-Modell"

A.3.4 Package - Lauf

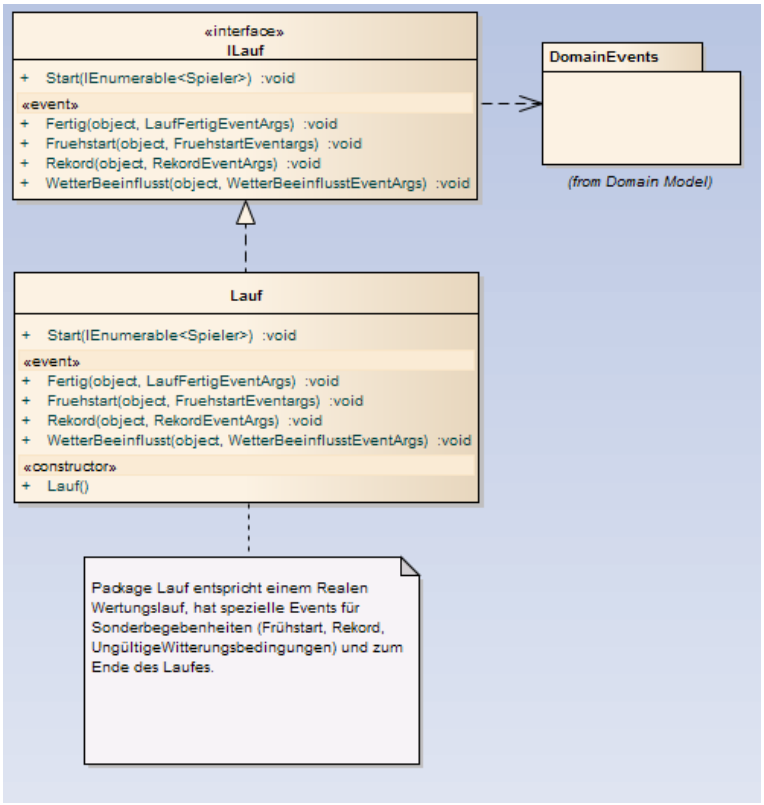

Abbildung A.12: Mögliche Realisierung des Packages "Lauf aus einem Domänen-Modell"

Literaturverzeichnis

[Fowler03] Martin Fowler: *Patterns für Enterprise Application-Architekturen*. mitp-verlag, 2003, ISBN 978-3-8266-1378-4

[Fowler99] Martin Fowler: *Analysemuster Wiederverwendbare Objektmodelle*. Addison-Wesley, 1999, ISBN 3-8273-1434-8

[Richter] Jeffrey Richter: *Microsoft .NET Framwork-Programmierung mit C#*. Microsoft Press, 2006, ISBN 3-86063-984-6

[Buschmann] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal: *Pattern-Oriented Software Architekture: A System of Patterns.* Volume 1 der POSA-Serie. Wiley, 1996, ISBN 978-0-471-95869-7

[Starke] Gernot Starke, Peter Hruschka: *Software-Architektur kompakt - angemessen und zielorientiert*, Spektrum Akademischer Verlag, 2009, ISBN 978-3-8274-2093-0

[Larman] Craig Larman *UML 2 und Patterns angewendet Objektorientierte Softwareentwicklung*. mitp-verlag, 2005, ISBN 978-3-8266-1453-8

[Gamma] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Entwurfsmuster > Elemente wiederverwendbarer objektorientierter Software*, Addison-Wesley, 2011, ISBN 978-3-8273-3043-7

[Pavlik] Frank Pavlik *Warum IT-Projekte häufig scheitern?* `http://www.domendos.com/fachlektuere/fachartikel/artikel/scheitern-von-it-projekten/` abgerufen am 23.10.2011 um 13:11Uhr

[Brooks] Frederick Brooks *The Mythical Man-Month*, Addison-Wesley, 1995, ISBN 0-201-83595-9

[Posch] Torsten Posch, Klaus Birken, Michael Gerdom *Basiswissen Softwarearchitektur*, dpunkt.verlag, 2011, ISBN 978-3-89864-736-6

[Schwichtenberg] Dr. Holger Schwichtenberg *Microsoft .NET 3.5 Crashkurs*, Microsoft Press, 2008, ISBN 978-3-86645-512-2

Glossar

Binding-By-Name Zwei Komponenten werden entkoppelt, indem eine nur den Namen einer aufzurufenden Methode, einer Eigenschaft oder eines Commands kennt.

CodeBehind Bei deklarativen Ansätzen (wie XAML) wird empfohlen, den statischen Anteil deklarativ zu beschreiben, dynamische Anteile sollen in einer Quelltextdatei (der Code-Behind-Datei) implementiert werden. Der Compiler erzeugt aus der deklarativen Beschreibung eine partielle Klasse zu der Klasse aus der Code-Behind-Datei.Bei deklarativen Ansätzen (wie XAML) wird empfohlen, den statischen Anteil deklarativ zu beschreiben, dynamische Anteile sollen in einer Quelltextdatei (der Code-Behind-Datei) implementiert werden. Der Compiler erzeugt aus der deklarativen Beschreibung eine partielle Klasse zu der Klasse aus der Code-Behind-Datei..

Common Language Runtime Schicht des .NET die IL-Assembler in Maschinen Code übersetzt..

Cross-Cutting Concern Bezeichnet in der Softwareentwicklung Aspekte die nicht modularisiert werden können, da sie im ganzen Quelltext verstreut sind. Meist handelt es sich um nichtfunktionale Anforderungen (wie Sicherheit).

Dependency Injection ist eine Anwendung von Inversion of Control, speziell im Bereich der Objekterzeugung. Objekte werden nicht mehr direkt erzeugt und initialisiert, dies geschieht von dem Verwender der Funktionalität. In vielen wird diese Aufgabe durch eine über Meta konfigurierbare Komponente erfüllt. Martin Fowler unterscheidet drei Typen von Dependency Injection:

- Constructor Injection – die zu verwendenden Objekte werden beim Initialisieren des Objektes über den Konstruktor mitgegeben.
- Setter Injection – die zu verwendenden Objekte können über Eigenschaften der Klasse gesetzt werden.
- Interface Injection – die zu verwendenen Objekte sind Teil der Methodensignatur und somit in der Schnittstelle definiert.

Weitere Informationen finden Sie unter `http://www.martinfowler.com/articles/injection.html`.

First-Mover Englischer Begriff für ein Pionierunternehmen, dieses Unternehmen hat ein Produkt in den Markt eingeführt, zudem es kein technisch vergleichbares Konkurrenzprodukt gibt.

Garbage Collector Der Garbage Collector von .NET Framework verwaltet die Belegung und Freigabe von Arbeitsspeicher für die Anwendung. Bei jedem Erstellen eines neuen Objekts belegt die Common Language Runtime (CLR) Speicher für das Objekt aus dem verwalteten Heap. Solange ein Adressbereich im verwalteten Heap verfügbar ist, reserviert die Laufzeit Arbeitsspeicher für neue Objekte. Arbeitsspeicher ist jedoch nicht unendlich verfügbar. Möglicherweise muss mithilfe der Garbage Collection Arbeitsspeicher freigegeben werden. Das Optimierungsmodul der Garbage Collection bestimmt den besten Zeitpunkt für das Einsammeln anhand der erfolgten Speicherbelegungen. Beim Einsammeln durch die Garbage Collection wird nach Objekten im verwalteten Heap gesucht, die nicht mehr von der Anwendung verwendet werden. Anschließend werden die für das Freigeben des Arbeitsspeichers erforderlichen Operationen ausgeführt.

Hook Beschreibt in der Softwareentwicklung eine Möglichkeit um fremden Maschinencode in eine Anwendung zu integrieren. Der fremde Maschinencode kann den Workflow des Programmes beeinflussen, Ergebnisse modifizieren oder Ereignisse (wie Windows-Nachrichten) abfangen. Das bekannteste Beispiel sind systemglobale Key- oder Maus-Hooks, welche es allen Programmen ermöglichen auf Mausbewegung oder Tastaturanschläge zu reagieren, auch wenn die Programme keinen Fokus haben.

Implicit Invocation Ist eng verwandt mit Inversion of Control. Bei diesem Architekturmuster werden Events ausgelöst, anstatt Methoden direkt auszuführen. Die auszuführende Methode registriert sich innerhalb der Klasse für das Event. Dies hat den Vorteil, dass sich auch andere Komponenten auf das Event anmelden können und beim Eintreten des Ereignisses tätig werden.

Inversion of Control Beschreibt ein Architekturmuster zum Entwerfen von Frameworks. Ein Unterschied zwischen einem Framework und einer Bibliothek besteht darin, dass in einem Framework Methoden vom Verwender des Framework aufgerufen werden. Dies verändert den Kontrollfluss, bei einer klassischen Bibliothek, ruft der Verwender eine Methode auf, deren Funktionalität wird abgearbeitet und die Methode kehrt zum Aufrufer zurück. Die Steuerung des Kontrollflusses liegt also in der Hand des Verwenders der Bibliothek. Ein gutes Beispiel ist das .NET Framework, der Verwender kann Methoden über Events an Ereignisse binden, dass .NET Framework entscheidet wann es die Funktionalität des Verwenders aufruft. Weitere Informationen finden Sie unter `http://martinfowler.com/bliki/InversionOfControl.html`.

Managed Heap Das .NET Framework verwendet einen verwalteten Heap. Dieser wird benutzt wenn Referenz Datentypen (Klassen / Arrays) angelegt werden. Wert Datentypen (Int, Boolean, Char, Strukturen) werden nicht auf dem Heap, sondern auf dem Stack angelegt. Der verwaltete Heap hat als Aufgaben:

- Ungenutzte Objekte als "Müll" zukennzeichnen

- Ungenutzte Speicherblöcke freigeben
- Heap defragmentierung

.

Post-Mortem Lateinisch für "nach dem Tod", bezeichnet in der Softwareentwicklung den Zustand nach dem Beenden des Programmes.

proprietär Abgeleitet vom lateinischen Wort für "Eigentümlichkeit", bezeichnet in der Softwareentwicklung nicht freie Komponenten oder Komponenten mit geschlossener Architektur.

Quelltextgenerierung Quelltextgenerierung bietet die Möglichkeit zur Laufzeit des Programmes Quelltext in IL-Code kompilieren. Dafür wird dynamisch im Speicher ein neues Assembly generiert und vom Programm sofort verwendet. Ein wesentlicher Nachteil besteht darin, dass mehrfaches kompilieren identischer Quelltexte, mehrere Assemblies im Speicher erzeugt. Sollte die Erzeugung nicht in einer AppDomain erfolgen, können die generierten Assemblies nicht wieder entladen werden. In der .NET Klassenbibliothek befinden sich die notwendigen Klassen im Namensraum System.CodeDom.

Reflection Reflektion liest aus den Metadaten der Objekte die Informationen über die Objekte selbst bzw. deren Typen aus, d.h. die Objekte reflektieren über sich selbst. Die Metadaten beinhalten Informationen über Methoden, Eigenschaften, Events usw. der Objekttypen.

Responsibility-Driven Design (RDD) Ist eine allgemeine Methapher, es geht darum Softwareobjekte wie Personen (inkl. derer Verantwortungen) in Zusammenarbeit mit anderen Personen zubetrachten, zur Lösung von Aufgaben zubetrachten. Durch die Sicht des RDD soll der Objektorientierte Entwurf wie eine *Gemeinschaft kollaborierender verwantwortlicher Objekte* wirken.

Stack Der Stapelspeicher arbeitet nach dem LIFO-Prinzip (Last in First Out). Stacks dienen dazu Rücksprungadressen, lokale Variablen o.ä. transient zu speichern. Um Daten auf dem Stack abzulegen dient der Befehl PUSH um das letzte Datum vom Stack zu entfernen benutzt man POP.

Sunny-Day-Szenarien Beschreibt die Obermenge von Anwendungsfällen (use cases), bei denen alles funktioniert.

transparent Definition nach [Fowler03, Seite 31]:
In der Informatik hat sich für den englischen Terminus *opaque*, dt. eigentlich *undurchsichtig*, im Deutschen der Terminus *transparent* eingebürgert, obwohl er eigentlich das genaue Gegenteil bedeutet. Der Informatiker sagt *transparent*, wenn er *nicht sichtbar, nicht merklich* meint. Beispiel: "Die Authentisierung verläuft für den Benutzer transparent" bedeutet für den Benutzer nicht merklich, nicht spürbar. Opaque als Anglizismus ist nicht etabliert.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Mittweida, 06. 01 2012